Tuto Jeu PHP OO + Documenter son code

1 Ma première classe (Conventions et syntaxe de base)

Pour débuter ce cours et poser les bases de la syntaxe en PHP OO, je ne vais pas travailler sur un exemple lié à la navigation web (trop complexe et abstrait pour l'instant) mais plutôt partir sur l'élaboration d'un jeu avec des personnages (combat, puissance, barre de vie etc.)

En POO, de manière conventionnelle, chaque classe (moule) à son propre fichier.

Ainsi, dans mon index.php, je ne vais pas coder de classe, mais les appeler selon les besoins. En l'occurrence ici, je vais appeler ma classe Personnage

```php
<?php

require 'personnage.php';
```

Et dans la mesure où il n'existe pas encore, je le crée dans la foulée

Remarque: par convention, le nom de la classe débutera par une majuscule, et le nom du fichier sera similaire (même nom, majuscule non obligatoire)

Autre convention; utiliser le camelCase pour les noms composés de propriétés et méthodes

```php
<?php

class Personnage{

}
```

Avec ses deux fichiers, je peux désormais instancier ma classe Personnage en créant un objet issu d'elle. Dans index.php, je vais utiliser cette syntaxe

```php
$magicien = new Personnage();
```

$magicien est une instance de ma classe Personnage.

D'ailleurs, si je fais un var_dump de cet objet, je vais en avoir la confirmation sur mon navigateur

```php
var_dump($magicien);
```

object(Personnage)#1 (0) { }

C'est bien un objet !

Pour rendre mon personnage intéressant, je vais lui donner des propriétés (ou attributs) et la première concernera sa barre de vie.

Une propriété/attribut est l'équivalent de la variable lorsqu'on code en Orienté Objet. Et pour la déclarer, je vais devoir lui donner une accessibilité, une visibilité. Ici, elle va être public (je reviendrai plus tard sur ce concept accessibilité/visibilité)

```
class Personnage{

  public $barreDeVie
 = 100;

}
```

Comme pour une variable, elle débute par le signe $

Si je rafraîchis mon navigateur, cette information apparaitra dans le var_dump

```
object(Personnage)#1 (1) {
  ["barreDeVie"]=>
  int(100)
}
```

Je lui donne une seconde caractéristique, sa puissance en attaque

```
public $attaque = 20;
```

Tout va bien, à son tour elle sera prise en compte dans le var_dump

Jusqu'à présent j'ai déclaré deux propriétés a ma classe. Je peux lui donner aussi des méthodes.

Méthode étant l'équivalent de fonction en POO.

Je vais déclarer une méthode pour son cri de guerre lorsqu'il part au combat

```
public function incantation(){
    echo "Abracadabra";
  }
```

Cette méthode ne sera pas récupérable directement dans le var_dump.

Si je veux la vérifier, je vais devoir utiliser une syntaxe particulière avec le signe →

```
var_dump($magicien->incantation())
```

Le signe → permet de pointer vers la méthode. Si j'avais voulu cibler seulement une des deux propriétés, j'aurai écrit de la même manière

```
var_dump($magicien->attaque);
```

Noter cependant que dans le cas de la méthode, son nom est suivi des parenthèses (), alors que dans le cas de l 'attribut, non (c'est bien entendu logique)

Je vais à présent déclarer une autre méthode, plus intéressante et plus complexe à exploiter.

Elle va concerner la capacité des personnages à régénérer leur barre de vie après en avoir perdu durant un combat, par exemple

Pour les besoins de la démonstration, j'abaisse la barre de vie en dur dans le code (je la baisse à 80)

```
public $barreDeVie = 80;
```

dans index.php, c'est bon, elle est à 80.

```
var_dump($magicien->barreDeVie);
```

Mon problème maintenant, c'est comment la remonter à 100. J'ai une mauvaise solution pour l'instant, c'est en dur dans index.php

En dessous de l'instanciation de mon personnage, je peux lui donner une valeur ainsi

```
$magicien = new Personnage();
$magicien->barreDeVie = 100;

echo '<pre>'; var_dump($magicien->barreDeVie); echo '</pre>';
```

Mais ce n'est pas une bonne manière de faire.

Mon objectif est de le faire à partir d'une fonction. Pas de coder en dur

Dans ma fonction regenerer(), je vais utiliser une variable très utile $this, qui me permet de pointer vers l'objet courant. Et je vais coder ceci

```
public function regenerer(){
    $this->barreDeVie = 100;
  }
```

Si je vérifie l'affichage de mon navigateur, la valeur est encore à 80.

Avec la syntaxe ci-dessous, je vais pouvoir la récupérer dans mon index.php

```
$magicien->regenerer();
```

Je vérifie, c'est bon, elle est bien passée à 100.

Si maintenant je décide d'instancier un nouvel objet de ma classe et que je vérifie sa barre de vie

```
$sorcier = new Personnage();
echo '<pre>'; var_dump($sorcier->barreDeVie); echo '</pre>';
```

Cette dernière est à 80. Logique, je n'ai pas demandé a ce nouveau personnage de se régénérer!

Je peux donc conclure que chaque objet est indépendant d'un autre, instancié de la même classe

En écrivant

```
$magicien->regenerer();
```

Je rentre dans la méthode regenerer() et le $this à l'intérieur pointera vers l'objet courant $magicien.

Pour qu'il puisse pointer vers $sorcier, il aurait fallu ecrire

```php
$sorcier->regenerer();
```

Et ainsi $this peut pointer vers lui et lui donner la nouvelle valeur de 100.

A ma classe, je vais ajouter une propriété pour donner un nom à mes futurs personnages

Je l'initialise, sans pour autant lui affecter une valeur (qui sera donnée plus tard, a chaque nouvelle instance (objet) de ma classe)

Voici a quoi doit ressembler le fichier de ma classe à ce stade du code

```php
<?php

class Personnage{

  // public $barreDeVie = 100;
  public $barreDeVie = 80;
  public $attaque = 20;
  public $nom;

  public function incantation(){
    echo "Abracadabra";
  }

  public function regenerer(){
    $this->barreDeVie = 100;
  }

}
```

Et mon index.php

```php
<?php

require 'Personnage.php';

$magicien = new Personnage();

$magicien->regenerer();

echo '<pre>'; var_dump($magicien->barreDeVie); echo '</pre>';

$sorcier = new Personnage();
// $sorcier->regenerer();
echo '<pre>'; var_dump($sorcier->barreDeVie); echo '</pre>';
```

Je peux néanmoins d'ors et déjà affecter un nom à mon magicien et à mon sorcier avec cette syntaxe

```php
$magicien->nom = "Merlin";
$sorcier->nom = "Gargamel";
```

```
echo '<pre>'; var_dump($magicien); echo '</pre>';
echo '<pre>'; var_dump($sorcier); echo '</pre>';
```

L'affichage de mon var_dump donnera ce résultat

```
object(Personnage)#1 (3) {
  ["barreDeVie"]=>
  int(100)
  ["attaque"]=>
  int(20)
  ["nom"]=>
  string(6) "Merlin"
}

object(Personnage)#2 (3) {
  ["barreDeVie"]=>
  int(80)
  ["attaque"]=>
  int(20)
  ["nom"]=>
  string(8) "Gargamel"
}
```

Noter que j'aurai pu ne pas initialiser ma propriété $nom dans ma classe et la déclarer en lui affectant une valeur directement dans index.php, cela aurait fonctionné (faites le test)

Mais cette pratique est déconseillée pour des raisons de clarté, de maintenance du code.

Les propriétés d'une classe doivent être déclarées en premier en son sein

---

A présent je vais améliorer mon code de manière a pouvoir donner un nom à mon personnage/objet, au moment où je l'instancie. C'est a dire qu'au lieu d'écrire cela une fois mon objet crée

```
$magicien->nom = "Merlin";
```

J'aimerais avoir une syntaxe qui s'approche de qlq chose comme cela

```
$elfe = new Personnage("Tauriel");
```

Comme je l'ai dit, au moment de l'instanciation

Pour cela, je vais devoir coder un constructeur dans ma classe Personnage et pour cela faire appel a une méthode __construct()

```
public function __construct($nom){

  }
```

Cette méthode, je lui donne en premier un paramètre qu'elle devra réceptionner $nom

Et pour passer ce paramètre à la propriété $nom (de ma classe) je vais a nouveau utiliser $this→ pour pointer vers l'objet en cours (celui que j'instancie)

```
public function __construct($nom){
```

```
    $this->nom = $nom;
  }
```

détail de cette syntaxe: $this pointe donc vers mon nouvel objet $elfe. Le nom juste après la flèche fait référence à la propriété $nom alors que le $nom qui suit le signe d'affection égal se réfère au paramètre reçu.

Le nommage de mon paramètre n'a aucune importance, j'aurai pu ecrire à la place

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

Et c'est déjà bien plus clair. Mais par convention je vais rester sur la première syntaxe (par convention, l'argument porte le nom de la propriété à laquelle il doit affecter une valeur)

Autre moyen de vérifier qui pointe vers quoi, vous pouvez cliquer sur un des noms et vous verrez à qui il se réfère (Visual Studio Code vous les mettra en surbrillance)

A noter, désormais avec ce constructeur qui indique que l'objet doit prendre un paramètre au moment de son instanciation, il faudra leur faire automatiquement, sinon erreur PHP

Fatal error: Uncaught ArgumentCountError: Too few arguments to function Personnage::__construct()

J' adapte donc mon code ainsi

```
$magicien = new Personnage("Merlin");

$magicien->regenerer();

// echo '<pre>'; var_dump($magicien->barreDeVie); echo '</pre>';


$sorcier = new Personnage("Gargamel");
// $sorcier->regenerer();
// echo '<pre>'; var_dump($sorcier->barreDeVie); echo '</pre>';


// $magicien->nom = "Merlin";
// $sorcier->nom = "Gargamel";

echo '<pre>'; var_dump($magicien); echo '</pre>';
echo '<pre>'; var_dump($sorcier); echo '</pre>';

$elfe = new Personnage("Tauriel");
echo '<pre>'; var_dump($elfe); echo '</pre>';
```

Codons maintenant une méthode qui vérifie si un personnage est mort ou non.

Je peux le faire de deux manières, une syntaxe plutôt longue mais qui aura le mérite d'être explicite, et une plus courte, moins fluide au débuter

```
public function mort(){
    if($this->barreDeVie == 0){
      return True;
    }else{
      return False;
    }
  }
```

Je vérifie dans un premier temps si ma propriété $barre_de_vie est == à 0 (et je pointe vers elle avec $this...si je ne le fais pas, si j'appelle directement $barre_de_vie, j'aurai une erreur PHP, undefined variable etc....

Si c'est le cas, elle me retourne le booléen True. Dans le cas contraire, False.

Une fois scriptée, j'applique cette méthode sur mon objet pour vérifier son état

```
var_dump($sorcier->mort());
```

La barre de vie de mon sorcier étant de 80, le var_dump me retournera False

Je peux modifier sa valeur à zéro

```
$sorcier->barreDeVie = 0;
var_dump($sorcier->mort());
```

Et effectivement, désormais c'est True qui sera retourné

La seconde syntaxe pour ma méthode mort() est

```
public function mort(){
    return $this->barreDeVie == 0;
  }
```

C'est la même condition, implicite => mort() est vérifié dans le cas où elle retourne $barre_de_vie == 0, sinon elle ne se vérifie pas elle retournera donc False

Et pour éviter qu'elle renvoie false en cas de valeur négative, il faudra mieux assurer avec un inférieur ou égal

```
return $this->barreDeVie <= 0;
```

Je vais à présent, pour continuer dans ma logique, permettre de régénérer un personnage en passant un paramètre à la méthode regenerer() dans index.php

```
$magicien->regenerer(10);
$sorcier->regenerer();
```

Je veux que si aucun paramètre n'est passé, la barre de vie remonte à 100 automatiquement

Au cas où une valeur est passée en paramètre, alors cette valeur sera additionnée à sa barre de vie actuelle

Voici la syntaxe, que je détaillerai juste après

```
public function regenerer($barreDeVie = null){
  if(is_null($barreDeVie)){
    $this->barreDeVie = 100;
  }else{
    $this->barreDeVie = $this->barreDeVie + $barreDeVie;
    // $this->barre_de_vie += $barre_de_vie;
  }

}
```

En premier, à ma méthode regenerer() je passe un paramètre par défaut $barreDeVie = null (au cas ou aucun paramètre ne soit renseigné). Ensuite, is_null($barreDeVie) est True (c'est à dire que aucune valeur n'a été envoyée en paramètre, alors ma propriété $barreDeVie (pointée par $this→) prendra la valeur de 100.

Si elle n'est pas nulle, qu'une valeur a été entrée, alors ma propriété $barreDeVie prendra sa propre valeur à laquelle on ajoute la valeur entrée en paramètre (en dessous, la même syntaxe, mais en forme contractée avec +=)

Et effectivement, la barre de vie de Merlin passera à 90 (ses 80 + 10 en paramètre) tandis que celle de Gargamel passera à 100, aucune valeur n'ayant été renseignée

---

Maintenant on va voir que je peux passer autre chose qu'une valeur en paramètre d'une fonction. Je peux lui passer par exemple un objet

Ça serait le cas si un de mes personnages pouvait désormais en attaquer un autre

```
$magicien->attaquer($sorcier);
```

Et la méthode attaquer serait scriptée ainsi dans ma classe

```
public function attaquer($cible){

  }
```

$cible étant son paramètre (en l'occurrence l'objet $sorcier ici) et je peux le vérifier rapidement un faisant un var_dump de $cible

```
public function attaquer($cible){
    echo '<pre>'; var_dump($cible); echo '</pre>';
  }
```

Et effectivement

```
object(Personnage)#2 (3) {
  ["barreDeVie"]=>
```

```
  int(0)
  ["attaque"]=>
  int(20)
  ["nom"]=>
  string(8) "Gargamel"
}
```

$cible est bien mon second objet de la classe Personnage, qui a pour nom Gargamel

Je peux même faire une autre vérification pour être sur que c'est bien l'objet $sorcier qui va être impacté.

```
public function attaquer($cible){
    $cible→barreDeVie = 20;
    // echo '<pre>'; var_dump($cible); echo '</pre>';
  }
```

J'affecte une nouvelle valeur à la barre de vie de ma cible directement dans la fonction attaquer()

Résultat, le var_dump rafraîchis de $sorcier indique bien que sa barre de vie indique bien 20 alors que celle des autres est restée telle que

```
object(Personnage)#2 (3) {
  ["barreDeVie"]=>
  int(20)
  ["attaque"]=>
  int(20)
  ["nom"]=>
  string(8) "Gargamel"
}
```

C'est donc bien mon objet qui est ciblé et non un double crée indépendant de l'objet d'origine.

Maintenant, je veux faire en sorte que lorsque un personnage est attaqué, il se voit retiré de sa barre de vie autant de points que vaut l'attaque de son assaillant

Je vais dans un premier temps schématiser pour visualiser la future syntaxe

```
public function attaquer($cible){

    // $this = assaillant
    // $cible = attaqué

    // attaqué.barreDeVie -= assaillant.attaque
  }
```

Je sais que $this pointe vers l'objet courant (et dans le cas présent c'est $magicien). Je sais aussi que $cible représente l'objet attaqué (c'est ici $sorcier)

Il me faudra donc impacter la barre de vie de ma cible en lui retirant autant de points que compte la force d'attaque de l'objet courant.

En code, cela donnera qlq chose de plutôt simple

```
public function attaquer($cible){
    $cible->barreDeVie -= $this->attaque;
```

```
  }
```

Je vérifie avec le var_dump ; c'est bon, la vie de Gargamel est bien passée à 80 (auparavant à 100 du fait qu'il ai été régénéré sans valeur en paramètre (donc à 100))

Il me reste une dernière chose à faire, c'est voir si cette attaque a été fatale (si Gargamel est décédé suite à l'attaque de Merlin).

Et pour cela je vais coder cette condition dans mon index.php

```
if($sorcier->mort()){
  echo $sorcier->nom . " a succombé à l'attaque de " . $magicien->nom;
}else{
  echo $sorcier->nom . " est toujours vivant. Il lui reste " . $sorcier->barreDeVie
. " points en barre de vie !";
}
```

Ça fonctionne, j'ai bien mon second message avec en prime la valeur de sa barre de vie

Gargamel est toujours vivant. Il lui reste 80 points en barre de vie !

Vérifiez que cela fonctionne bien dans l'autre cas (en modifiant au-dessus de cette même condition la valeur de la barre de vie)

```
$sorcier->barreDeVie = 0;
```

Ça fonctionne toujours

Gargamel a succombé à l'attaque de Merlin

2 Visibilité (public, private) et getters/setters

2-1 visibilité public vs private

Je reprends mes deux fichiers précédents (nettoyés des commentaires) pour comprendre cette notion de niveau de visibilité

Le niveau public veut dire que toutes mes propriétés et leurs valeurs sont visibles (et récupérables) non seulement à l'intérieur de ma classe (je pointe dessus avec $this), mais aussi en dehors.

Je l'ai bien vu dans le chapitre précédent, je pouvais récupérer la valeur de la propriété nom directement dans mon index.php

```
echo $sorcier->nom . " a succombé à l'attaque de " . $magicien->nom;
```

Je vais à présent modifier dans le fichier de ma classe Personnage la visibilité de $nom. Je vais la passer de public en private...je teste

```
private $nom;
```

Si je rafraîchis ma page, j'ai désormais une erreur PHP qui arrête mon script

Fatal error: Uncaught Error: Cannot access private property Personnage::$nom

2-2 Getter

Pour remédier à cela je vais coder une méthode (getNom) qui me permettra de récupérer cette valeur malgré sa visibilité private

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

Je rappelle qu'avec private, je peux toujours y accéder avec $this. Cette valeur est toujours visible dans ma classe (j'ajoute juste désormais return pour pouvoir en disposer à l'extérieur de ma méthode)

Par contre, je dois adapter ma syntaxe dans index.php. Je ne pointerai désormais plus vers ma propriété $sorcier->nom, mais vers la méthode $sorcier->getNom()

```
// echo $sorcier->nom . " a succombé à l'attaque de " . $magicien->nom;
  echo $sorcier->getNom() . " a succombé à l'attaque de " . $magicien->getNom();
```

Ça fonctionne à nouveau.

Il existe un autre niveau de visibilité qui est protected. Il se situe entre public, le plus permissif et private, le plus restrictif. Mais nous y reviendrons plus tard lors du chapitre sur la notion d'héritage. Pour l'instant, on retient juste que protected est équivalent à private

La question que l'on est en droit de se poser, par rapport à tout ce que l'on vient de voir, c'est pourquoi ne pas tout garder en visibilité public ?

Ces niveaux de visibilité, d'accessibilité permettent de protéger le code des interventions extérieures, non pas malveillantes nécessairement, mais surtout maladroites, involontaires ou incompétentes.

Image de la voiture

Le jour ou je décide de construire le moule, ma classe voiture.

Je vais donner l'autorisation au conducteur d'avoir accès aux propriétés volant, boite d'embrayage, code de la route...mais pas à la propriété moteur. Il n'a pas cette compétence. Il n'en a même pas besoin d'ailleurs.

Dans le meilleur des cas, je l'encombre avec qlq chose qui lui est inutile pour conduire la voiture. Mais je prends le risque qu'il modifie les réglages du moteur, et casse toute la voiture.

Les niveaux de visibilité permettent donc de mieux gérer les interventions extérieures.

Retour à mon code et à mon cas présent, mon jeu

Précédemment, j'avais fait en sorte que même si par hasard la barre de vie prenait une valeur négative, la méthode mort() retournait tout de même True (et pas seulement si elle est strictement égale à zéro)

A présent, je vais scripter une méthode pour empêcher qu'une valeur négative soit affectée à la propriété $barreDeVie

Et je vais lui donner un niveau de visibilité private pour qu'elle ne puisse être accessible qu'à l'intérieur de ma classe. Je ne veux pas qu'elle puisse être visible en dehors. C'est l'objectif

```
private function pasNegative(){

  }
```

Et je vais l'utiliser dans ma fonction attaquer() pour éviter, par exemple, qu'une barre de vie restante de 10, soit amputée d'une attaque de 20 et se retrouve à -10

```
public function attaquer($cible){

    $cible->barre_de_vie -= $this->attaque;
    $cible->pasNegative();
  }
```

retour à ma fonction pasNegative() pour lui donner son comportement dans son bloc d'instructions

```
private function pasNegative(){
    if($this->barreDeVie < 0){
      $this->barreDeVie = 0;
    }
  }
```

Je dis que si la barre de vie de l'objet courant prend une valeur inférieure à zéro, alors je lui donne la valeur de zéro.

Et surtout, je la veux private car je n'en aurai besoin que dans le cadre de ma classe. Je n'aurais pas à l'utiliser ailleurs. Je ne veux d'ailleurs pas.

Concrètement, le jour ou un autre développeur aura accès à mes fichiers, il saura tout de suite ce qu'il pourra modifier, et ce qui doit être protéger pour le bon fonctionnement du code.

Par convention, toutes nos propriétés auront une visibilité private et pour donner la possibilité de récupérer leur valeur, nous créerons comme tout à l'heure des getter

Je vais dans la foulée modifier mon code pour respecter cette méthodologie

```
// public $barreDeVie = 80;
private $barreDeVie = 80;

// public $attaque = 20;
private $attaque = 20;
```

```php
private $nom;

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

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

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

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

Et je modifie en conséquence (comme tout à l'heure) ma syntaxe dans index.php. Je dois désorais pointer vers une méthode et non une propriété

```php
echo $sorcier->getNom() . " est toujours vivant. Il lui reste " . $sorcier->getBarreDeVie() . " points en barre de vie !";
```

2-3 Setter

Nous venons donc de voir les getter, qui permettent de récupérer une valeur privée.

Il existe les setter, qui eux vont permettre d'injecter une nouvelle valeur (tant que cette valeur est autorisée)

Leur syntaxe par ailleurs va beaucoup être proche de celle d'un constructeur

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

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

Le $this désignant l'objet en cours et pointant vers la propriété, lui affectant la valeur entrée en paramètre tel que ci dessous

```php
$magicien->setNom("Houdini");
```

Un setter est lié à une visibilité restreinte telle que private dans la mesure où il impose un protocole pour pouvoir injecter une nouvelle valeur (ou modifier une existante). Il vient donc en appoint de cette logique de protection en amont.

Je vais ainsi pouvoir controler la nature de la valeur entrée.

Si je demande un numéro de téléphone, c'est dans le setter que je contrôlerai qu'on ne m'injecte pas autre chose que huit chiffres. Si la condition codée est respectée, alors la valeur entrée en paramètre deviendra la nouvelle valeur de la propriété de mon objet courant !

A la fin de ce chapitre, voici à quoi devront ressembler mes index.php

```php
<?php

require 'Personnage.php';

$magicien = new Personnage("Merlin");

$magicien->regenerer(10);

$sorcier = new Personnage("Gargamel");

$sorcier->regenerer();

$elfe = new Personnage("Tauriel");

$magicien->attaquer($sorcier);

$magicien->setNom("Houdini");

echo '<pre>'; var_dump($magicien); echo '</pre>';
echo '<pre>'; var_dump($sorcier); echo '</pre>';
echo '<pre>'; var_dump($elfe); echo '</pre>';

if($sorcier->mort()){
  // echo $sorcier->nom . " a succombé à l'attaque de " . $magicien->nom;
  echo $sorcier->getNom() . " a succombé à l'attaque de " . $magicien->getNom();
}else{
  // echo $sorcier->nom . " est toujours vivant. Il lui reste " . $sorcier-
>barre_de_vie . " points en barre de vie !";
  echo $sorcier->getNom() . " est toujours vivant. Il lui reste " . $sorcier-
>getBarre_de_vie() . " points en barre de vie !";
}
```

Et Personnage.php

```php
<?php

class Personnage{

// public $barre_de_vie = 80;
private $barre_de_vie = 80;
```

```php
// public $attaque = 20;
private $attaque = 20;

private $nom;

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

public function setBarreDeVie($barre_de_vie){
  $this->barre_de_vie = $barre_de_vie;
}

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

public function setAttaque($attaque){
  $this->attaque = $attaque;
}

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

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

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

public function incantation(){
  echo "Abracadabra";
}

public function regenerer($barre_de_vie = null){
  if(is_null($barre_de_vie)){
    $this->barre_de_vie = 100;
  }else{
    $this->barre_de_vie = $this->barre_de_vie + $barre_de_vie;
  }

}

private function pasNegative(){
  if($this->barre_de_vie < 0){
    $this->barre_de_vie = 0;
  }
```

```
}

public function mort(){
  return $this->barre_de_vie <= 0;
}

public function attaquer($cible){

  $cible->barre_de_vie -= $this->attaque;
  $cible->pasNegative();
}

}
```

3 Documenter ses classes

Il s'agit non seulement de mettre de la documentation à la disposition du développeur qui lirait notre code, mais aussi d'envoyer ces mêmes informations aux générateurs de documentation comme aux IDE (ici Visual Studio Code)

Pour cela, il faut respecter différentes règles syntaxiques comme méthodologiques

En tout premier, je vais documenter ma classe, tout en haut de mon fichiers

```
/**
 * Class Personnage
 * permet de faire combattre des personnages entre eux
 */

class Personnage{
```

Je vais ensuite en faire de même pour mes attributs comme méthodes

```
/**
 * @var integer indique la vie disponible pour chaque personnage au début
 */
private $barreDeVie = 80;
```

Avec @var j'indique que c'est un attribut, ensuite son type, puis sa fonction/utilité

```
/**
 * @var integer indique la puissance disponible pour chaque personnage au début
 */
private $attaque = 20;

/**
 * @var string indique le nom de chaque objet/personnage
 */
private $nom;
```

Pour le constructeur, la syntaxe différera légèrement (@param au lieur de @var), mais tout en respectant la même logique.

```
/**
 * @param $nom string
 * permet de donner un nom à chaque personnage au moment ou on instancie l'objet
 */

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

J'indique donc que cette méthode prend en argument $nom qui sera de type string

En fait, PHP n'étant pas un langage typé (c'est a dire qu'au moment de la déclaration de l'attribut je n'indique pas son type), cette documentation permet justement de renseigner cette information. Je n'ai désormais plus à chercher dans le code qu'elle type de valeur elle attend. C'est indiqué en amont !

Je vais continuer à documenter mes autres méthodes, avec deux cas de figure différents

```
/**
 * @param $barreDeVie integer
 * permet de modifier la barre de vie du personnage
 */

public function setBarreDeVie($barreDeVie){
  $this->barreDeVie = $barreDeVie;
}

/**
 * @return integer
 */

public function getBarreDeVie(){
  return $this->barreDeVie;
}
```

Dans le cas de mon setter, j'indique le nom de mon paramètre ainsi que le type de valeur qu'il attend.

En revanche, mon getter ne prend aucun paramètre, donc je n'indique aucun @param. Par contre je vais préciser le type de la valeur qu'il retourne avec @return

Voici la suite de mes méthodes désormais documentées

```
/**
 * @param $attaque integer
 * permet de modifier la valeur de l'attaque
 */
public function setAttaque($attaque){
  $this->attaque = $attaque;
```

```php
}

/**
 * @return integer
 */
public function getAttaque(){
  return $this->attaque;
}

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

/**
 * @return string
 */
public function getNom(){
  return $this->nom;
}

/**
 * @return string
 */
public function incantation(){
  return "Abracadabra";
}

/**
 * @param $barreDeVie integer
 * permet de modifier/remonter la barre de vie à 100 si le personnage a une barre de vie = 0 (mort), ou de la rehausser de la valeur renseignée en parametre (si pas renseignée, est équivalente à null)
 */
public function regenerer($barreDeVie = null){
  if(is_null($barreDeVie)){
    $this->barreDeVie = 100;
  }else{
    $this->barreDeVie = $this->barreDeVie + $barreDeVie;
  }

}

/**
 * condition qui évite que la barre de vie prenne une valeur négative
 */
private function pasNegative(){
  if($this->barreDeVie < 0){
    $this->barreDeVie = 0;
  }
}
```

```php
/**
 * @return boolean
 * vérifie si un personnage est mort ou non
 */
public function mort(){
  return $this->barreDeVie <= 0;
}

/**
 * @param $cible string
 * soustrait à la barre de vie de la cible la valeur de l'attaque qu'il subit, tout en empêchant que cette valeur (barre de vie) ne devienne négative
 */
public function attaquer($cible){

  $cible->barreDeVie -= $this->attaque;
  $cible->pasNegative();
}
```

Documentez vos attributs et méthodes au fur et a mesure !