Victor Thuillier

PROGRAMMEZ EN ORIENTÉ OBJET EN

# PHP

2e édition

OPENCLASSROOMS

EYROLLES

PROGRAMMEZ EN ORIENTÉ OBJET EN

# PHP

2e édition

Difficile aujourd'hui de passer à côté de la programmation orientée objet ! La quasi-totalité des outils créés actuellement pour les développeurs PHP utilise cette façon de programmer. Il est donc indispensable de savoir ce que c'est et comment s'en servir. Que vous ayez déjà quelques notions sur ce concept ou que vous soyez complètement novice en la matière, cet ouvrage est fait pour vous !

## QU'ALLEZ-VOUS APPRENDRE ?

**Théorie : les bases de la POO**

- Introduction à la POO
- L'utilisation des classes
- L'opérateur de résolution de portée
- La manipulation des données stockées
- TP : minijeu de combat
- L'héritage
- TP : personnages spécialisés
- Les méthodes magiques

**Théorie : techniques avancées**

- Les objets en profondeur
- Les interfaces
- Les exceptions
- Les traits
- L'API de réflexivité
- UML : présentation
- UML : modélisation des classes
- Les design patterns
- TP : un système de news
- Les générateurs
- Les closures

**Pratique : réalisation d'un site web**

- Description de l'application
- Développement de la bibliothèque
- Le front-end
- Le back-end
- Gérer les formulaires

## À PROPOS DE L'AUTEUR

Passionné par le Web, Victor Thuillier apprend grâce au site OpenClassrooms à créer son premier site à l'âge de 12 ans. Voulant aller plus loin, il décide d'approfondir ses connaissances dans le domaine, et plus particulièrement sur le langage PHP. Il découvre la programmation orientée objet à l'âge de 14 ans et s'en sert pour réaliser de nombreux sites Internet. Fort de son expérience, il rédige ensuite ce cours dans le but d'aider ceux qui, comme lui auparavant, souhaitent découvrir cette façon de programmer.

## L'ESPRIT D'OPENCLASSROOMS

Des cours ouverts, riches et vivants, conçus pour tous les niveaux et accessibles à tous gratuitement sur notre plate-forme d'e-éducation : www.openclassrooms.com. Vous y vivrez une véritable expérience communautaire de l'apprentissage, permettant à chacun d'apprendre avec le soutien et l'aide des autres étudiants sur les forums. Vous profiterez des cours disponibles partout, tout le temps : sur le Web, en PDF, en eBook, en vidéo…

www.editions-eyrolles.com

PROGRAMMEZ EN ORIENTÉ OBJET EN

# PHP

DANS LA MÊME COLLECTION

J. Pardanaud, S. de la Marck. – **Découvrez le langage JavaScript.**
N°14399, 2017, 478 pages.

A. Bacco. – **Développez votre site web avec le framework Symfony3.**
N°14403, 2016, 536 pages.

M. Chavelli. – **Découvrez le framework PHP Laravel.**
N°14398, 2016, 336 pages.

R. De Visscher. – **Découvrez le langage Swift.**
N°14397, 2016, 128 pages.

M. Lorant. – **Développez votre site web avec le framework Django.**
N°21626, 2015, 285 pages.

E. Lalitte. – **Apprenez le fonctionnement des réseaux TCP/IP.**
N°21623, 2015, 300 pages.

M. Nebra, M. Schaller. – **Programmez avec le langage C++.**
N°21622, 2015, 674 pages.

SUR LE MÊME THÈME

H. Bersini. – **La programmation orientée objet.**
N°67399, 7e édition, 2017, 696 pages.

P. Martin, J. Pauli, C. Pierre de Geyer, É. Daspet. – **PHP 7 avancé.**
N°14357, 2016, 732 pages.

E. Biernat, M. Lutz. – **Data science : fondamentaux et études de cas.**
N°14243, 2015, 312 pages.

B. Philibert. – **Bootstrap 3 : le framework 100 % web design.**
N°14132, 2015, 318 pages.

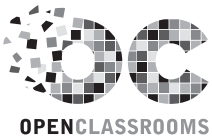
C. Delannoy. – **Le guide complet du langage C.**
N°14012, 2014, 844 pages.

Victor Thuillier

# PROGRAMMEZ EN ORIENTÉ OBJET EN

# PHP

2e édition




EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com



ISBN : 978-2-212-14472-7

# Introduction

Difficile aujourd'hui de passer à côté de la programmation orientée objet ! Peut-être n'en avez-vous d'ailleurs jamais entendu parler ; si c'est le cas, nous allons remédier à cela immédiatement ! La quasi-totalité des outils créés actuellement pour les développeurs PHP utilise cette façon de programmer, il est donc indispensable de savoir ce que c'est et comment s'en servir. Que vous ayez déjà quelques notions sur ce concept ou que vous soyez complètement novice en la matière, cet ouvrage est fait pour vous ! Nous partirons de zéro et découvrirons ensemble toute la puissance de cette nouvelle façon de développer.

Cet ouvrage a été pensé pour être suivi par quiconque possédant des connaissances en PHP. Nous commencerons par poser les bases de ce nouveau concept avec quelques exercices et travaux pratiques pour assimiler ces notions. Après avoir découvert des notions plus avancées, nous réaliserons ensemble un site web complet de A à Z ! Vous vous rendrez compte au fil de la lecture de cet ouvrage de la façon dont la programmation orientée objet vous permettra d'organiser vos projets de manière plus cohérente, assurera une maintenance facilitée et vous permettra de partager vos codes de façon plus aisée !

Mais avant de vous lancer dans ce (très) vaste domaine, il est indispensable, voire obligatoire :

- d'être à l'aise avec PHP et sa syntaxe ; si ce n'est pas le cas, suivez le cours « Concevez votre site web avec PHP et MySQL » *(http://openclassrooms.com/courses/concevez-votre-site-web-avec-php-et-mysql)* ;
- d'avoir bien pratiqué ;
- d'être patient ;
- d'avoir au moins PHP 5.4 sur son serveur.

Si vous avez déjà pratiqué d'autres langages permettant la programmation orientée objet, c'est un gros plus, surtout si vous savez programmer en Java, car PHP a principalement tiré son modèle objet de ce langage.

On y va ?

# Table des matières

## Pratique : réalisation d'un site web 321

Première partie

# Théorie : les bases de la POO

# 1 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 cet ouvrage par 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. Imaginons par exemple que 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 !

## Naissance de 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 livre 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 ces 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 ;
- acquérir 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 qu'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, 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 ?

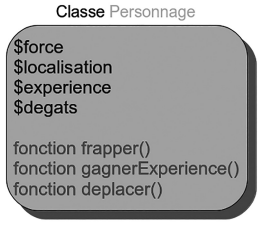
Une classe est une entité regroupant des variables et des fonctions. Chacune de ces fonctions aura accès aux variables de cette entité. Dans le cas du personnage, nous aurons une fonction `frapper()`. Cette fonction devra simplement modifier la variable `$degats` du personnage en fonction de la variable `$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 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 citées précédemment : c'est la base de tout objet `Personnage`. Chaque instance de cette classe possédera ainsi toutes ces variables et fonctions (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 qu'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 permet d'effectuer des actions sur l'avion. C'est l'*interface* de l'avion. Le pilote se moque de savoir 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 trifouiller dedans. 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.

Il est temps à présent de 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. Mais auparavant, je vais vous donner la syntaxe de base de toute classe en 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, celui-ci sera représenté par une variable, et c'est à partir de celle-ci qu'on pourra modifier l'objet, appeler des méthodes, etc. Vous allez donc dire à PHP « dans cet objet, donne-moi cet attribut » ou « dans cet objet, appelle cette méthode » : c'est ça, appeler des attributs ou méthodes depuis l'extérieur de l'objet.

Le second, `private`, impose quelques restrictions. On aura accès aux attributs et méthodes *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 ceci :

```
<?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
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 écrire de `private $_attribut = 1 + 1` ou bien `private $_attribut = 'Hello ' . 'world !'`.

## Création de méthodes

Pour la déclaration de méthodes, il suffit de faire précéder la visibilité de la méthode du mot-clé `function`. Les types de visibilité des méthodes sont les mêmes que ceux des 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
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 précède 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.

# 2 L'utilisation des classes

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

Ce chapitre sera aussi l'occasion d'approfondir un peu le développement de nos classes : actuellement, la classe qu'on a créée contient des méthodes vides, chose un peu inutile vous avouerez. Pas de panique, nous allons nous 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
$perso = new Personnage;
```

Ainsi, `$perso` sera un objet de type `Personnage`. On dit qu'on *instancie* la classe `Personnage`, c'est-à-dire qu'on crée une instance de la classe `Personnage`.

### Appeler les méthodes de l'objet

Pour appeler l'une des méthodes 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** qu'on veut utiliser. Dans l'exemple précédent, cet objet aurait été `$perso`. À droite de l'opérateur, on spécifie le nom de la méthode qu'on veut invoquer.

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

```
<?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 dit par ailleurs 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
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
class Personnage
{
  private $_experience;

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

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

Oui, mais voilà : comment accéder à l'attribut `$_experience` dans notre méthode ? C'est là qu'intervient la pseudo-variable `$this`.

Dans notre script, `$perso` représente l'objet. Il est donc facile d'appeler une méthode à partir de cette variable. Mais dans notre méthode, nous n'avons pas cette variable pour accéder à notre attribut `$_experience` pour le modifier ! Du moins, c'est ce que vous croyez. En fait, un paramètre représentant l'objet est passé implicitement à chaque méthode de la classe. Regardez plutôt cet exemple :

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

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

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

Commentons le contenu de la méthode `afficherExperience`. 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()`. Elle devra ajouter 1 à l'attribut `$_experience`. Je suis sûr que vous pouvez y arriver !

Voici la correction :

```
<?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 que nous en implémentions 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
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
// 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
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.

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

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 :

```
<?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 ceci :

```
<?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
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.

## 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 le 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
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 qu'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 peut 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
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
$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 second.

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

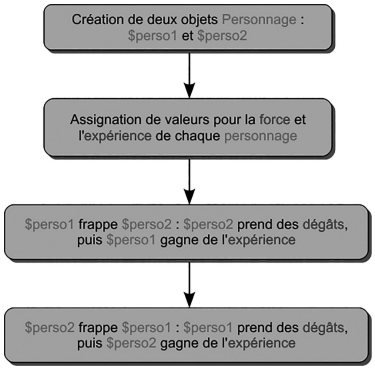
```
<?php
$perso1 = new Personnage; // Un premier personnage
$perso2 = new Personnage; // Un second personnage

$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 ! Le schéma suivant résume 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 connaître 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* ; par conséquent, aucune valeur ne doit être retournée, même si cela peut ne générer 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
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 après la création
                                    // d'un 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 qu'on vient de créer. Il faudra donc lui spécifier ceci en paramètre :

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

La figure suivante présente le résultat affiché en sortie :


Résultat affiché par le script

Notez quelque chose d'important dans la classe : dans le constructeur, les valeurs sont initialisées en appelant les mutateurs correspondants. 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 et 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'autochargement 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'aurai qu'à inclure ce fichier :

```
<?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 inutiles si vous ne vous servez pas de toutes vos classes. Et c'est là qu'intervient l'autochargement 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
function chargerClasse($classe)
{
  require $classe . '.php'; // On inclut la classe correspondant au
                            // paramètre passé.
}
```

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

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

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

Et là… nous obtenons une erreur fatale ! La classe n'a pas été trouvée, elle n'a donc pas été chargée… Normal quand on y réfléchit ! PHP ne sait pas qu'il doit appeler cette fonction lorsqu'on essaye d'instancier une classe non déclarée. On va donc utiliser la

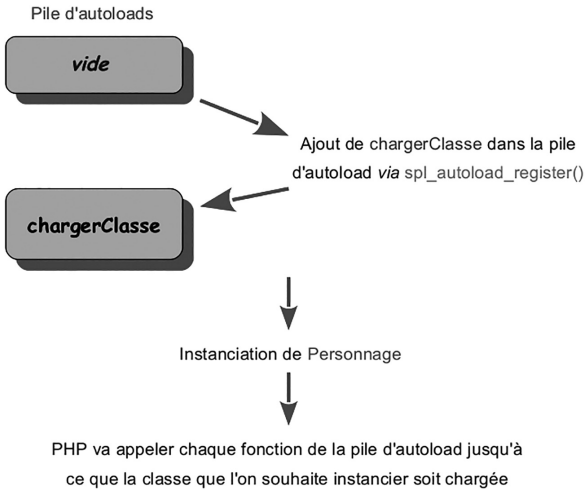
fonction `spl_autoload_register` en spécifiant en premier paramètre le nom de la fonction à charger :

```
<?php
function chargerClasse($classe)
{
  require $classe . '.php'; // On inclut la classe correspondant 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 autochargement 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 lorsqu'on essaye d'instancier une classe non déclarée. Nous avons donc ajouté ici notre fonction à la pile d'autoloads, afin qu'elle soit appelée à chaque fois qu'on essaye d'instancier une classe non déclarée.

La figure suivante montre schématiquement ce qu'il 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 l'attribut ou à la 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 les déclarant) grâce à l'autochargement de classe (utilisation de `spl_autoload_register`).

# 3 L'opérateur de résolution de portée

L'opérateur de résolution de portée (`::`), appelé « doubles 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, bravo !).

## 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 exemple de code muet :

```
<?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 trois valeurs possibles :

- `20`, qui signifie que le personnage aura une faible force ;