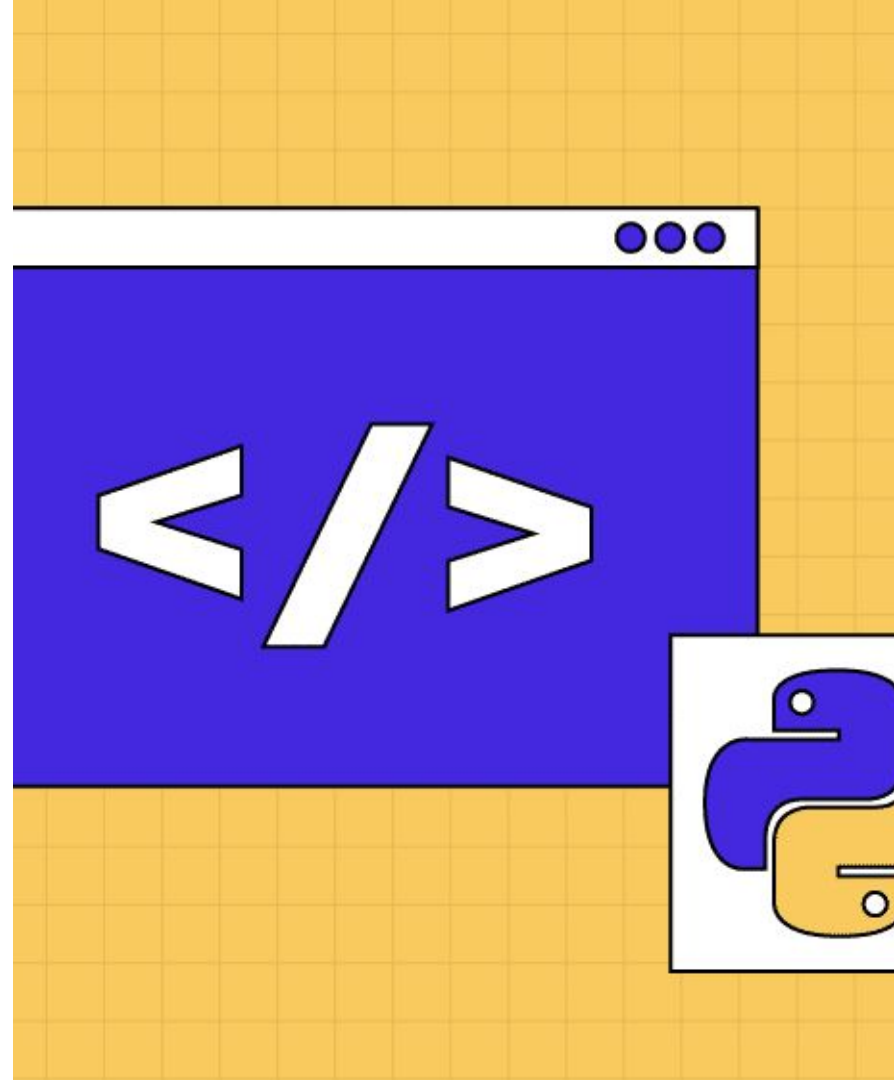





















Programmation Orientée Objet (POO) en PHP

Valentin RIBEZZI

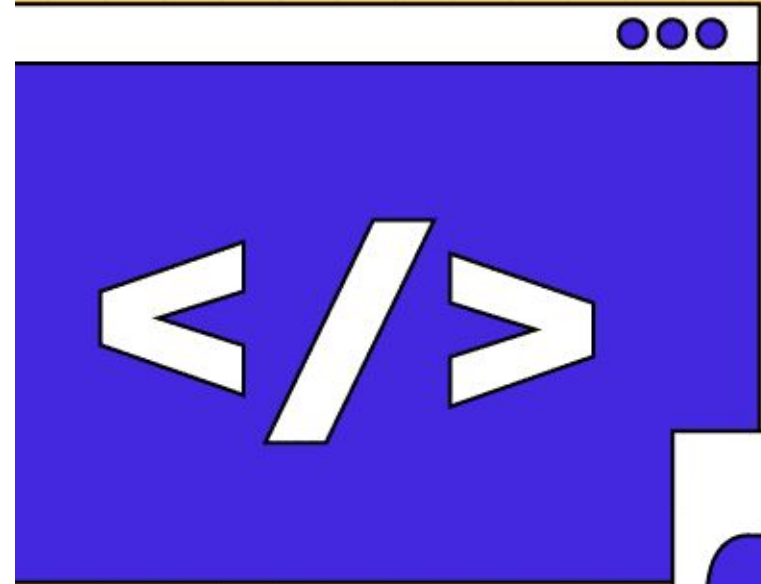


Nous allons créer un **MINI** Elden Ring
en PHP en utilisant la **P**rogrammation
Orientée **O**bjet.

- ▼  config
 -  config.php
- ▼  public
 - ▼  assets
 - ▼  styles
- ▼  src
 - >  Controller
 - ▼  Core
 -  Controller.php
 -  Database.php
 -  Request.php
 -  Response.php
 -  Router.php
 - >  Entity
 - >  Repository
 - >  Service
 - >  View
 -  index.php
 -  README.md

Sommaire

- I. Introduction à la POO
- II. Concepts fondamentaux de la POO
- III. Encapsulation et visibilité
- IV. Héritage et polymorphisme
- V. Abstraction et interfaces

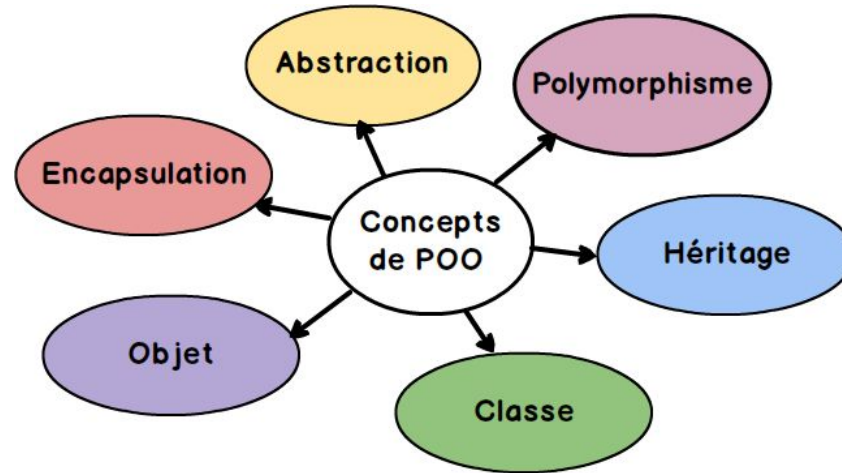


01

Programmation Orientée Objet (POO) en PHP

Introduction à la POO

Introduction à la POO



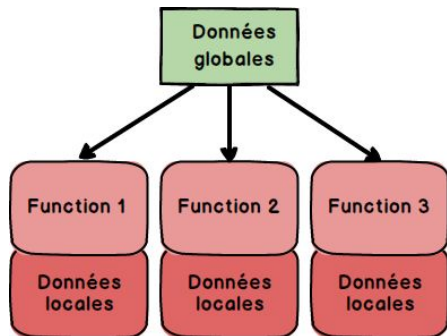
La **Programmation orientée objet (POO)** est une manière de résoudre un problème. C'est un paradigme, modèle de référence, une façon d'aborder un problème, une manière de penser et une concrétisation d'une philosophie de programmation.

Introduction à la POO

Jusqu'ici comment avons-nous l'habitude de programmer ? Et quelle est la structure de nos codes ?

Introduction à la POO

Procédure procédurale



La programmation procédurale est un paradigme de programmation organisé autour de procédures ou de sous-programmes. Le paradigme se concentre sur l'exécution séquentielle et la logique de programmation linéaire.

La programmation procédurale définit des procédures ou des fonctions pour des tâches spécifiques. Il utilise des variables, des boucles et des instructions conditionnelles pour contrôler le flux d'exécution.

Introduction à la POO

Les différences entre le procédurale et la POO

Aspect	Programmation Orientée Objet	Programmation procédurale
Organisation	Code structuré en objets regroupant données et comportements.	Code organisé en fonctions/procédures distinctes, avec une séparation nette entre fonctions et données.
Gestion des données	Données encapsulées dans des objets, accessibles via des méthodes (getters/setters).	Données souvent globales ou passées en paramètres entre fonctions.
Réutilisabilité	Réutilisation facilitée grâce à l'héritage et au polymorphisme ; possibilité d'extension sans modifier le code existant.	Réutilisation par appel de fonctions ; l'extension peut devenir complexe dans de gros projets.
Modélisation	Permet de modéliser des entités concrètes et leurs interactions (ex : utilisateurs, produits, personnages).	Moins intuitive pour représenter des entités complexes et leurs interactions.
Maintenance et évolutivité	Architecture modulaire facilitant la maintenance, les tests et l'évolution du code dans des projets de grande envergure.	Adaptée aux petits projets ou scripts simples, mais peut devenir difficile à maintenir sur le long terme.

Introduction à la POO



```
<?php
// Fonction qui calcule l'aire d'un rectangle
function rectangleArea($width, $height) {
    return $width * $height;
}

// Définition des dimensions du rectangle
$width = 5;
$height = 10;

// Calcul de l'aire
$area = rectangleArea($width, $height);

// Affichage du résultat
echo "L'aire du rectangle est de : $area";
?>
```

Introduction à la POO

```
<?php
// Déclaration de la classe Rectangle
class Rectangle {
    // Attributs privés pour encapsuler les données
    private $width;
    private $height;

    // Constructeur pour initialiser l'objet
    public function __construct($width, $height) {
        $this->width = $width;
        $this->height = $height;
    }

    // Méthode pour calculer l'aire du rectangle
    public function getArea() {
        return $this->width * $this->height;
    }
}

// Instanciation de l'objet Rectangle
$rectangle = new Rectangle(5, 10);

// Utilisation de la méthode de l'objet pour obtenir l'aire
echo "L'aire du rectangle est de : " . $rectangle->getArea();
?>
```

Introduction à la POO

Que faut-il retenir sur l'intérêt de la
Programmation **O**rientée **O**bjets ?

Utilisation de la notion d'objet (et d'instance)

Amélioration de la réutilisabilité et l'organisation du code

02

Programmation Orientée Objet (POO) en PHP

Concepts fondamentaux de la POO

Concepts fondamentaux de la POO

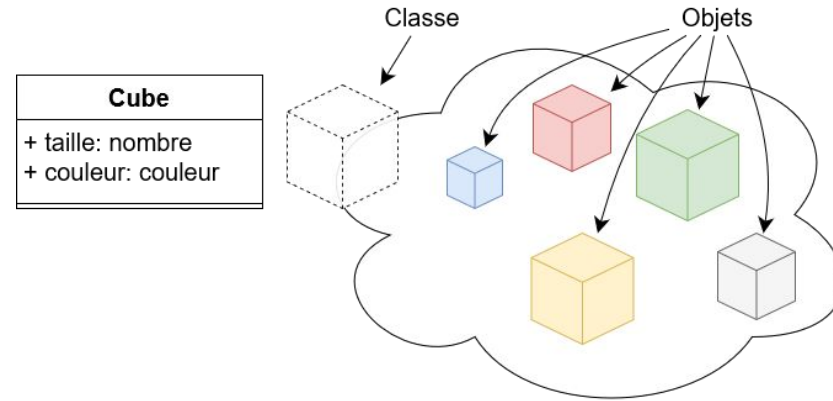
En POO, il existe **deux notions** indispensables pour comprendre son fonctionnement.



Concepts fondamentaux de la POO

Concrètement,
qu'est-ce qu'une **classe** ?

Concepts fondamentaux de la POO



Les classes sont des **moules**, des **patrons** qui permettent de créer des objets en série sur le même modèle. On peut se représenter une classe comme le schéma de construction ainsi que la liste des fonctionnalités d'un ensemble d'objets.

Concepts fondamentaux de la POO



```
class Character {  
    // Attributs, constructeur et méthodes  
}  
  
$hero = new Character();
```

*Pour cette classe, on créera le fichier
Character.php dans le dossier **src/Entity***


Concepts fondamentaux de la POO

Vu que l'on manipule des objets, des entités, chacun d'eux possède **des attributs spécifiques**.



```
class Character {  
    protected $name; // Attribut 1  
    protected $health; // Attribut 2  
    protected protected $attackPower; // Attribut 3  
}  
  
$hero = new Character();
```

Concepts fondamentaux de la POO

A code editor window with a blue border and a dark background. It contains a class definition for 'Character' and an object creation statement. The code is as follows:

```
class Character {  
    // Attributs, constructeur et méthodes  
}  
  
$hero = new Character();
```

Pour créer un ou des objets à partir d'une classe, on a besoin de le(s) construire.

Pour cela, nous allons créer une méthode à l'intérieur de notre classe pour effectuer cela, qui se nomme **Le constructeur**.

Concepts fondamentaux de la POO

```
class Character {  
    // Attributs, constructeur et méthodes  
}  
  
$hero = new Character();
```



```
class Character {  
    protected $name; // Attribut 1  
    protected $health; // Attribut 2  
    protected $attackPower; // Attribut 3  
  
    public function __construct(string $name, int $health, int $attackPower) {  
        $this->name = $name;  
        $this->health = $health;  
        $this->attackPower = $attackPower;  
    }  
}  
  
$hero = new Character("Chevalier Errant", 150, 25);
```

Notre méthode pour le constructeur (qui doit s'appeler comme cela) va prendre en paramètre les valeurs à définir pour les attributs de la classe.

Concepts fondamentaux de la POO

Comment faire pour que notre objet effectue des actions spécifiques ?

```
class Character {  
    protected $name; // Attribut 1  
    protected $health; // Attribut 2  
    protected protected $attackPower; // Attribut 3  
  
    public function __construct(string $name, int $health, int $attackPower) {  
        $this->name = $name;  
        $this->health = $health;  
        $this->attackPower = $attackPower;  
    }  
}  
  
$hero = new Character("Chevalier Errant", 150, 25);
```

Concepts fondamentaux de la POO

En POO, pour effectuer un ensemble d'actions ou d'instructions, nous avons **des méthodes**.

Si fonctions et méthodes peuvent prendre des entrées et renvoyer des sorties, les fonctions ne sont pas relatives à un objet. En fait, en programmation orientée objet pure, les fonctions n'existent pas puisque tout est objet, c'est-à-dire instance de classe.

Grâce à une méthode, on va pouvoir réaliser des **opérations** qui sont **spécifiques à un objet** : modifier ses attributs, les afficher, les retourner (ou les initialiser dans le cas de la méthode `__init__`), etc.

Concepts fondamentaux de la POO

Le premier genre de méthode que nous allons voir pour nos classes sont **les getters** et **les setters**.

```
class Character {  
    protected $name; // Attribut 1  
    protected $health; // Attribut 2  
    protected protected $attackPower; // Attribut 3  
  
    // Constructeur  
    public function __construct(string $name, int $health, int $attackPower) {  
        $this->name = $name;  
        $this->health = $health;  
        $this->attackPower = $attackPower;  
    }  
  
    // Getters  
    public function getName(): string {  
        return $this->name;  
    }  
  
    public function getHealth(): int {  
        return $this->health;  
    }  
  
    public function getAttackPower(): int {  
        return $this->attackPower;  
    }  
  
    // Setters  
    public function setHealth(int $health): void {  
        $this->health = $health;  
    }  
}  
  
// On crée nos objets  
$character1 = new Character("Chevalier Errant", 150, 25);  
$character2 = new Character("Seigneur des Ténèbres", 200, 30);
```

Il joue un rôle très important car ce sont eux qui vont permettre de modifier (proprement) les attributs de notre classe, mais également de les récupérer.

*On reparlera de leur importance dans l'**encapsulation**.*

Concepts fondamentaux de la POO

Nous allons maintenant créer une “vrai” méthode, qui va permettre à un personnage de lancer des attaques à d’autres.

Pour cela, nous allons créer la méthode **attack**.

*PS : On a aussi ajouté la méthode **displayStatus**, que l’on peut utiliser pour voir les points de vie et la puissance d’attaque de notre personnage.*

```
class Character {
    protected $name; // Attribut 1
    protected $health; // Attribut 2
    protected $attackPower; // Attribut 3

    // Constructeur
    public function __construct(string $name, int $health, int $attackPower) {
        $this->name = $name;
        $this->health = $health;
        $this->attackPower = $attackPower;
    }

    // Getters
    public function getName(): string {
        return $this->name;
    }

    public function getHealth(): int {
        return $this->health;
    }

    public function getAttackPower(): int {
        return $this->attackPower;
    }

    // Setters
    public function setHealth(int $health): void {
        $this->health = $health;
    }

    // Methodes
    public function attack(Character $target): void {
        $target->setHealth($target->getHealth() - $this->attackPower);
    }

    public function displayStatus(): void {
        echo "{$this->name} - Points de vie: {$this->health}, Attaque: {$this->attackPower}\n";
    }
}

// On crée nos objets
$character1 = new Character("Chevalier Errant", 150, 25);
$character2 = new Character("Seigneur des Ténèbres", 200, 30);
```


Concepts fondamentaux de la POO

En POO, il est possible de définir des objets comme des types pour des variables.

En l'occurrence ici, notre cible sera de type **Character** puisque c'est le type d'objet que l'on veut cibler.

```
// Méthodes
public function attack(Character $target): void {
    $target->setHealth($target->getHealth() - $this->attackPower);
}
```

1. Nous allons passer en paramètre de notre méthode une cible, qui sera un objet **Character**
2. Nous allons définir une nouvelle santé pour notre cible : Cette santé sera égale à sa santé actuelle mais la puissance d'attaque du personnage qui attaque la cible.

Concepts fondamentaux de la POO

```
class Character {
    protected $name; // Attribut 1
    protected $health; // Attribut 2
    protected $attackPower; // Attribut 3

    // Constructeur
    public function __construct(string $name, int $health, int $attackPower) {
        $this->name = $name;
        $this->health = $health;
        $this->attackPower = $attackPower;
    }

    // Getters
    public function getName(): string {
        return $this->name;
    }

    public function getHealth(): int {
        return $this->health;
    }

    public function getAttackPower(): int {
        return $this->attackPower;
    }

    // Setters
    public function setHealth(int $health): void {
        $this->health = $health;
    }

    // Methodes
    public function attack(Character $target): void {
        $target->setHealth($target->getHealth() - $this->attackPower);
    }

    public function displayStatus(): void {
        echo "{$this->name} - Points de vie: {$this->health}, Attaque: {$this->attackPower}\n";
    }
}

// On crée nos objets
$character1 = new Character("Chevalier Errant", 150, 25);
$character2 = new Character("Seigneur des Ténèbres", 200, 30);

// On appelle la méthode "attack" de notre classe
$character1->attack($character2);
```

03

Programmation Orientée Objet (POO) en PHP

Encapsulation et visibilité

Encapsulation et visibilité

On a vu que nos attributs, getters, setters et nos méthodes pouvaient être en **protected** ou en **public**.

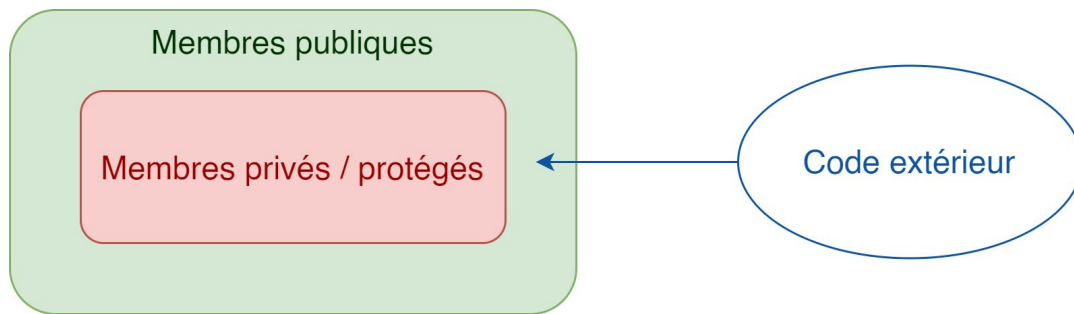
Mais qu'est-ce que cela signifie ?

Encapsulation et visibilité

En POO, il existe le principe d'**encapsulation**.

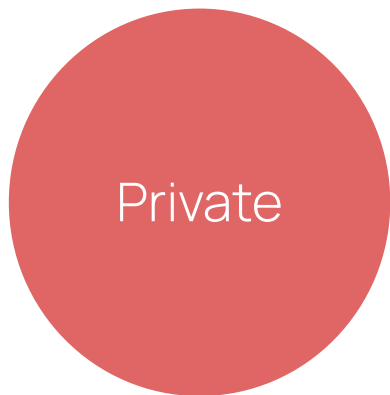
Le but est de cacher les détails de mise en oeuvre d'un objet.

Autrement dit, on veut réguler l'accès direct aux champs d'un objet depuis l'extérieur de celui-ci.



Encapsulation et visibilité

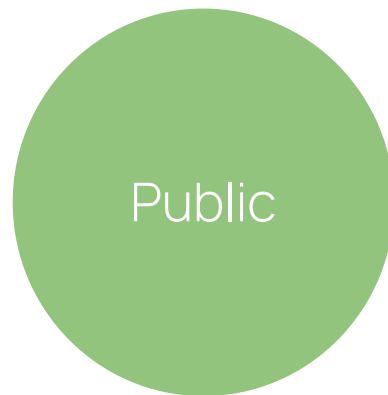
Il y a **3 niveaux** de visibilité pour une entité en POO.



Invisible en dehors
de la classe

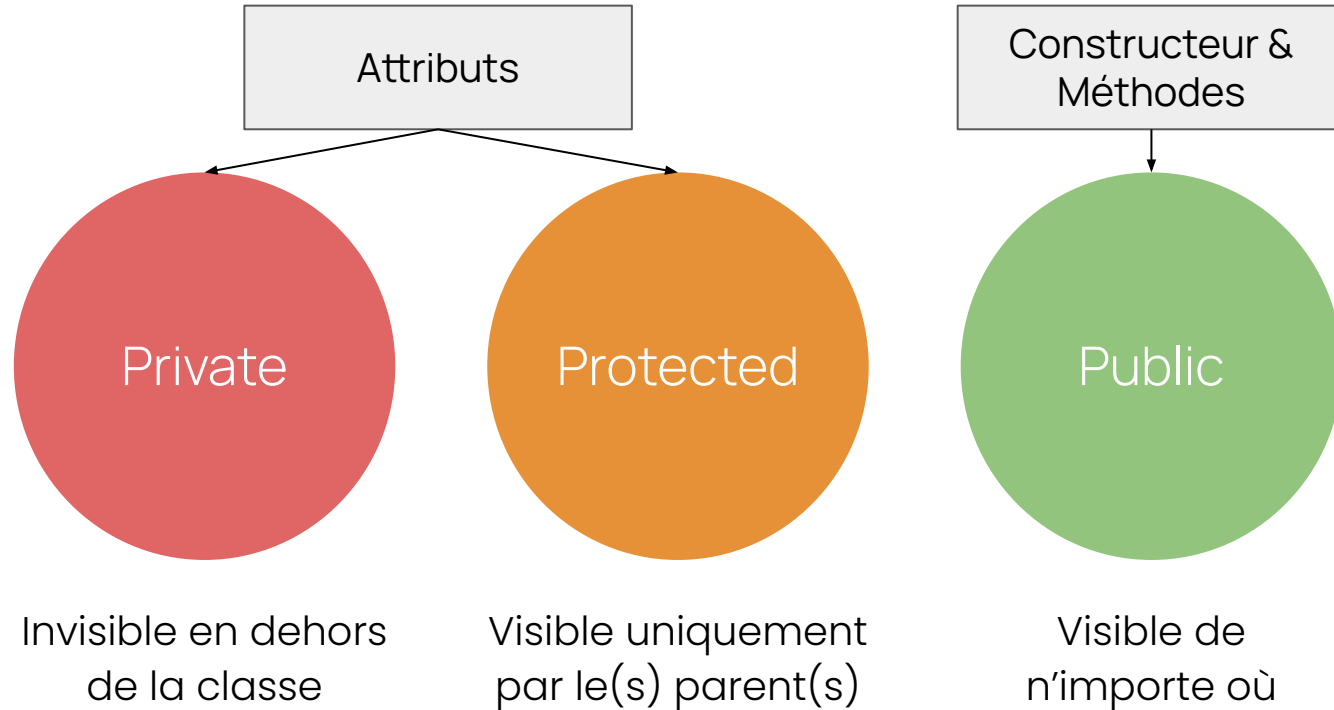


Visible uniquement
par le(s) parent(s)



Visible de
n'importe où

Encapsulation et visibilité



Encapsulation et visibilité

```
class Character {  
    // Attributs, méthodes, ....  
}
```

```
class Character {  
    protected $name; // Attribut 1  
    protected $health; // Attribut 2  
    protected $attackPower; // Attribut 3  
}
```

```
class Character {  
    protected $name; // Attribut 1  
    protected $health; // Attribut 2  
    protected $attackPower; // Attribut 3  
  
    // Constructeur  
    public function __construct(string $name, int $health, int $attackPower) {  
        $this->name = $name;  
        $this->health = $health;  
        $this->attackPower = $attackPower;  
    }  
}
```

```
class Character {  
    // Getters  
    public function getName(): string {  
        return $this->name;  
    }  
  
    public function getHealth(): int {  
        return $this->health;  
    }  
    public function getAttackPower(): int {  
        return $this->attackPower;  
    }  
  
    // Setters  
    public function setHealth(int $health): void {  
        $this->health = $health;  
    }  
  
    // Méthodes  
    public function attack(Character $target): void {  
        $target->setHealth($target->getHealth() - $this->attackPower);  
    }  
  
    public function displayStatus(): void {  
        echo "{$this->name} - Points de vie: {$this->health}, Attaque: {$this->attackPower}\n";  
    }  
}
```

```
// On crée nos objets  
$character1 = new Character("Chevalier Errant", 150, 25);  
$character2 = new Character("Seigneur des Ténèbres", 200, 30);  
  
// On appelle la méthode "attack" de notre classe  
$character1->attack($character2);
```


04

Programmation Orientée Objet (POO) en PHP

Héritage et polymorphisme

Héritage et polymorphisme

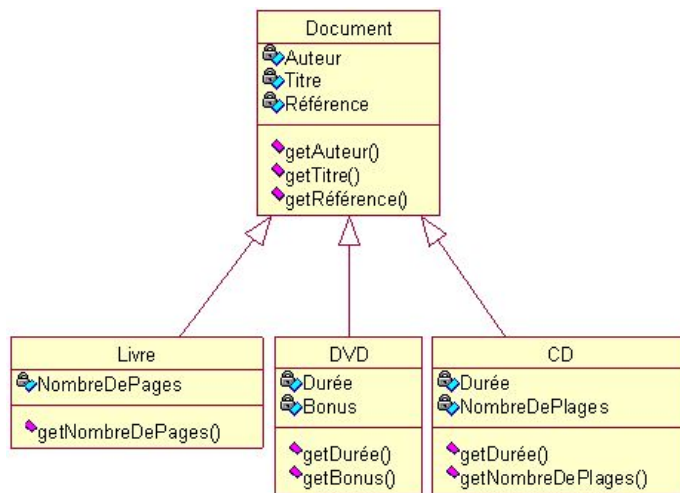
Jusqu'à maintenant, nous n'avons créé qu'une classe *générique* permettant de créer des personnages.

Nous allons à présent créer des "sous-classes" à partir de notre classe *générique*.

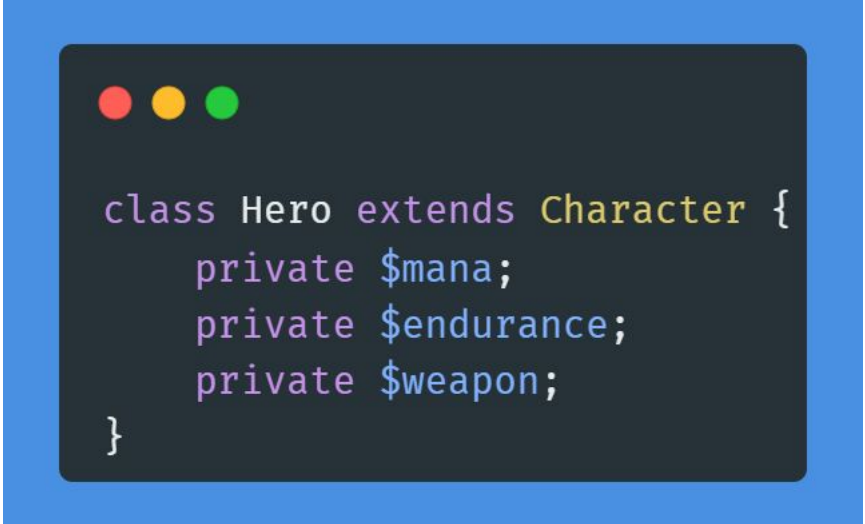
Cela se nomme de l'**héritage**.

Héritage et polymorphisme

L'héritage en programmation orientée objet permet de créer facilement des classes similaires à partir d'une autre classe. On parle alors de faire hériter une classe fille d'une classe mère.



Héritage et polymorphisme



```
class Hero extends Character {  
    private $mana;  
    private $endurance;  
    private $weapon;  
}
```

*Pour cette classe, on créera le fichier
Hero.php dans le dossier **src/Entity***

Héritage et polymorphisme

```
class Hero extends Character {  
    private $mana;  
    private $endurance;  
    private $weapon;  
  
    public function __construct(string $name, int $health, int $attackPower, int $mana = 100, int $endurance = 100) {  
        parent::__construct($name, $health, $attackPower);  
        $this->mana = $mana;  
        $this->endurance = $endurance;  
    }  
}
```

Notre constructeur va combiner le constructeur de notre **parent**, ainsi que la définition des valeurs des attributs de la classe **Hero**.

Pour le constructeur du parent, on va écrire **parent::__construct** (Comme si nous voulions créer un objet **Character**)

Héritage et polymorphisme

Lorsque l'on va créer un objet à l'aide de notre classe **Hero**, nous définirons à la fois les attributs de la classe **Hero**, mais aussi les attributs de la classe **Character**.



```
$character1 = new Character("Chevalier Errant", 150, 25);
```



```
$hero = new Hero("Chevalier Errant", 150, 25, 100, 50); // 100 mana, 50 endurance
```

Héritage et polymorphisme

L'utilisation de **max** pour les fonctions **decreaseMana** et **decreaseEndurance** permet de s'assurer que la valeur d'endurance ne descende jamais en dessous de 0.

```
class Hero extends Character {
    private $mana;
    private $endurance;
    private $weapon; // Arme équipée, instance de Weapon (optionnelle)

    public function __construct(string $name, int $health, int $attackPower, int $mana = 100, int $endurance = 100) {
        parent::__construct($name, $health, $attackPower);
        $this->mana = $mana;
        $this->endurance = $endurance;
    }

    // Getters
    public function getMana(): int {
        return $this->mana;
    }

    public function getEndurance(): int {
        return $this->endurance;
    }

    // Méthodes de modification
    public function increaseMana(int $amount): void {
        $this->mana += $amount;
    }

    public function decreaseMana(int $amount): void {
        $this->mana = max(0, $this->mana - $amount);
    }

    public function increaseEndurance(int $amount): void {
        $this->endurance += $amount;
    }

    public function decreaseEndurance(int $amount): void {
        $this->endurance = max(0, $this->endurance - $amount);
    }

    // Méthode pour équiper une arme
    public function equipWeapon(Weapon $weapon): void {
        $this->weapon = $weapon;
        echo "{$this->name} a équipé {$weapon->getName()} qui ajoute {$weapon->getDamageBonus()} points d'attaque.\n";
    }

    // La gestion des sorts sera réalisée dans les classes de sort (voir FireSpell)
}
```

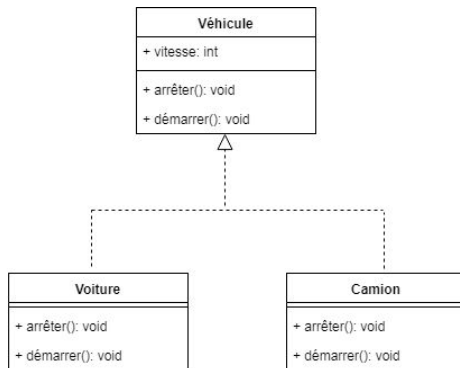
Héritage et polymorphisme

Une question peut alors se poser :

Que se passe t-il si deux Character
n'attaque pas de la même manière ?

Héritage et polymorphisme

C'est là qu'intervient le **polymorphisme**.



Le polymorphisme est un concept clé en programmation orientée objet (POO) qui permet aux objets de différentes classes d'utiliser des méthodes portant le même nom mais de se comporter de manière différente selon la classe à laquelle ils appartiennent.

Héritage et polymorphisme

Pour un **Character**, la méthode *attack* était ainsi :



```
public function attack(Character $target): void {  
    echo "{$this->name} attaque {$target->getName()} et inflige {$this->attackPower} points de dégâts.\n";  
    $target->setHealth($target->getHealth() - $this->attackPower);  
}
```

Héritage et polymorphisme

Dans **Hero**, nous redéfinissons la méthode comme ceci :

```
public function attack(Character $target): void {
    $attackCost = 10; // Coût en endurance pour une attaque

    if ($this->endurance < $attackCost) {
        echo "{$this->name} n'a pas assez d'endurance pour attaquer.\n";
        return;
    }

    $this->decreaseEndurance($attackCost);

    $totalDamage = $this->attackPower;
    if ($this->weapon) {
        $totalDamage += $this->weapon->getDamageBonus();
    }

    echo "{$this->name} attaque {$target->getName()} et inflige {$totalDamage} points de dégâts (coût
    endurance: {$attackCost}).\n";
    $target->setHealth($target->getHealth() - $totalDamage);
}
```

Héritage et polymorphisme

```
class Hero extends Character {
    private $mana;
    private $endurance;
    private $weapon; // Arme équipée, instance de weapon (optionnelle)

    public function __construct(string $name, int $health, int $attackPower, int $mana = 100, int $endurance = 100) {
        parent::__construct($name, $health, $attackPower);
        $this->mana = $mana;
        $this->endurance = $endurance;
    }

    // Getters
    public function getMana(): int {
        return $this->mana;
    }

    public function getEndurance(): int {
        return $this->endurance;
    }

    // Méthodes de modification
    public function increaseMana(int $amount): void {
        $this->mana += $amount;
    }

    public function decreaseMana(int $amount): void {
        $this->mana = max(0, $this->mana - $amount);
    }

    public function increaseEndurance(int $amount): void {
        $this->endurance += $amount;
    }

    public function decreaseEndurance(int $amount): void {
        $this->endurance = max(0, $this->endurance - $amount);
    }

    // Méthode pour équiper une arme
    public function equipWeapon(weapon $weapon): void {
        $this->weapon = $weapon;
        echo "{$this->name} a équipée {$weapon->getName()} qui ajoute {$weapon->getDamageBonus()} points d'attaque.\n";
    }

    // Redéfinition de attack() : vérifie l'endurance
    public function attack(Character $target): void {
        $attackCost = 10; // Coût en endurance pour une attaque

        if ($this->endurance < $attackCost) {
            echo "{$this->name} n'a pas assez d'endurance pour attaquer.\n";
            return;
        }

        $this->decreaseEndurance($attackCost);

        $totalDamage = $this->attackPower;
        if ($this->weapon) {
            $totalDamage += $this->weapon->getDamageBonus();
        }

        echo "{$this->name} attaque {$target->getName()} et inflige {$totalDamage} points de dégâts (coût endurance: {$attackCost}).\n";
        $target->setHealth($target->getHealth() - $totalDamage);
    }
}
```

Héritage et polymorphisme

Nous allons aussi créer la classe **Boss** qui hérite de **Character**.

```
class Boss extends Character {
    // Attaque spéciale : double les dégâts
    public function attack(Character $target): void {
        $damage = $this->attackPower * 2;
        echo "Le Boss {$this->name} attaque {$target->getName()} et inflige {$damage} points de dégâts (attaque spéciale) !\n";
        $target->setHealth($target->getHealth() - $damage);
    }
}
```

Pour cette classe, on créera le fichier **Boss.php** dans le dossier **src/Entity**

05

Programmation Orientée Objet (POO) en PHP

Abstraction et interfaces