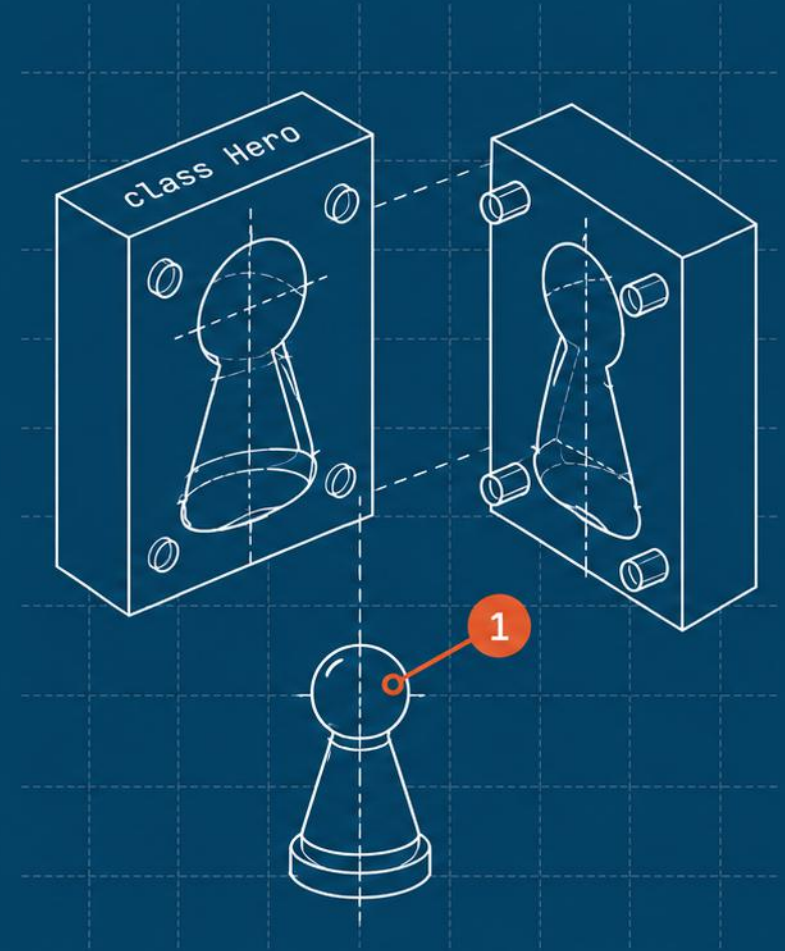


Bloc P00 • chapitre 1

Objets et classes

Un héros, en tableau puis en objet.



Un héros en tableau associatif

```
1 $hero = ['name' => 'Arthur', 'hp' => 10, 'maxHp' => 10, 'strength' => 2];
2
3 function takeDamage(array $hero, int $amount): array
4 {
5     $hero['hp'] = max(0, $hero['hp'] - $amount);
6
7     return $hero;
8 }
9
10 $hero = takeDamage($hero, 3);
11 $hero['hp'] = 9999;
12 $hero['maxHP'] = 12;
13 echo describe($hero);
```

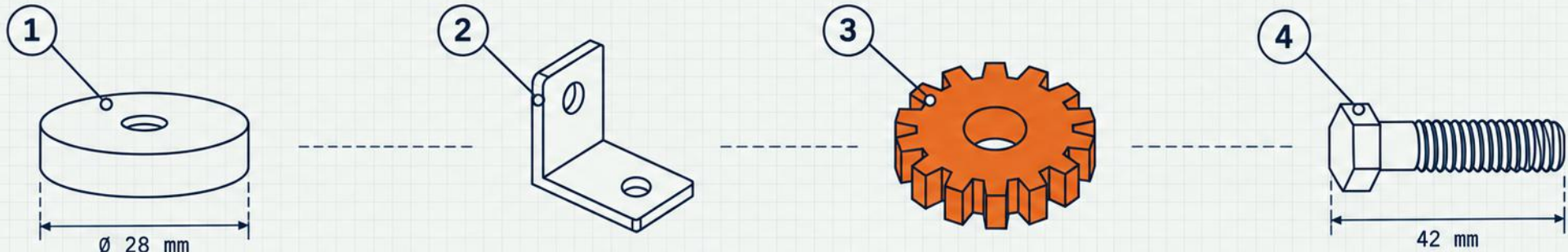
Arthur (9999/10 PV)

Un nom, des points de vie, une force. Avec ce que vous savez déjà : un tableau et des fonctions.

Le résultat est correct. Le code a quatre défauts.

1. **Données et fonctions séparées** : rien n'indique que `heal()` concerne le héros
2. **Réassigner à chaque appel** : oubliez `$hero =` et rien ne le signale
3. **Rien n'est protégé** : `$hero['hp'] = 9999`; passe
4. **Rien n'est vérifié** : `maxHP` crée une case au lieu de modifier `maxHp`

Ce sont les quatre que la POO corrige.

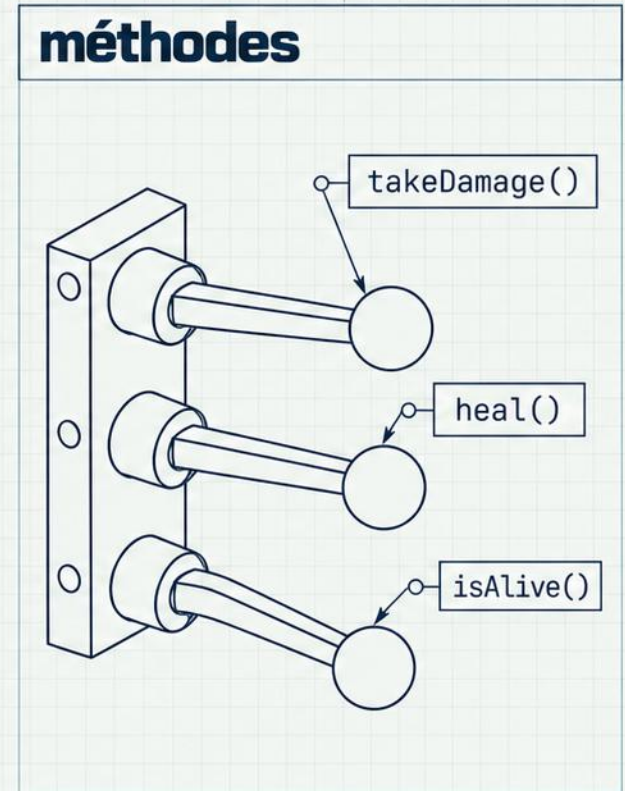
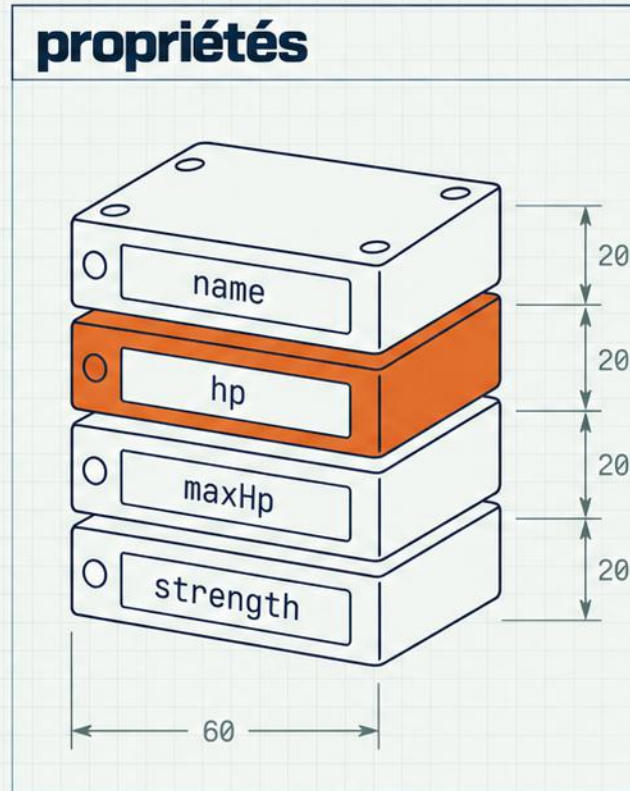


Propriété, méthode

Une **propriété** est une variable qui appartient à l'objet. Elle remplace une case du tableau : name, hp, maxHp, strength.

Une **méthode** est une fonction qui appartient à l'objet. Elle remplace une des fonctions : takeDamage(), heal(), isAlive().

Le reste n'est que de la syntaxe.



La même chose, en objet

```
1  <?php
2
3  declare(strict_types=1);
4
5  class Hero
6  {
7      public private(set) int $hp;
8      public private(set) int $maxHp;
9
10     public function __construct(
11         public readonly string $name,
12         int $maxHp = 10,
13         public readonly int $strength = 2,
14     ) {
15         $this->maxHp = $maxHp;
16         $this->hp = $maxHp;
17     }
```

1

état

Les quatre propriétés et le constructeur. Chaque mot nouveau est expliqué dans les slides qui suivent.

Les actions du jeu

```
18 public function takeDamage(int $amount): void
19 {
20     $this->hp = max(0, $this->hp - $amount);
21 }
22
23 public function heal(int $amount): void
24 {
25     $this->hp = min($this->maxHp, $this->hp + $amount);
26 }
27
28 public function isAlive(): bool
29 {
30     return $this->hp > 0;
31 }
32
33 public function __toString(): string
34 {
35     return sprintf('%s (%d/%d PV)', $this->name, $this->hp, $this->maxHp);
36 }
37 }
```

comportement

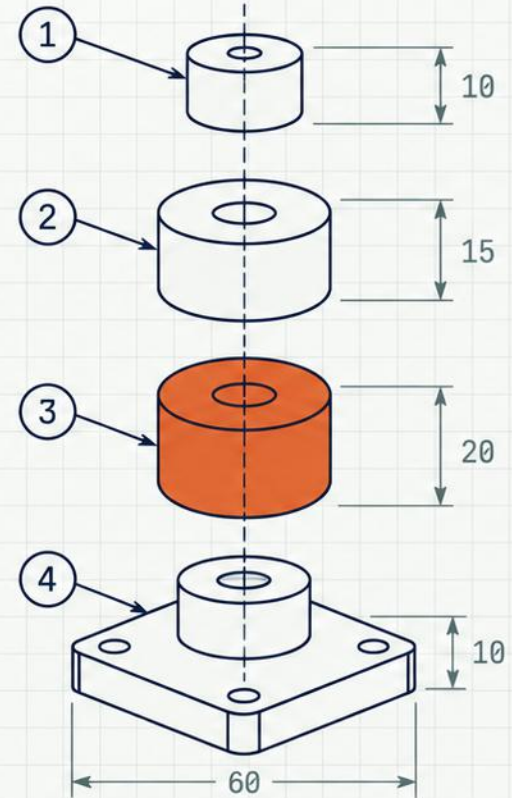
Suite du même fichier. Les trois fonctions du tableau sont devenues des méthodes, et la règle « jamais sous 0 » est écrite une seule fois.

\$hero est maintenant un seul bloc

```
1 $hero = new Hero('Arthur');
2 $hero->takeDamage(3);
3 $hero->heal(1);
4 echo $hero, PHP_EOL;
5
6 $hero->takeDamage(50);
7 echo $hero, PHP_EOL;
8 echo $hero->isAlive() ? 'vivant' : 'mort', PHP_EOL;
```

Arthur (8/10 PV)
Arthur (0/10 PV)
mort

Même résultat qu'avec le tableau. On ne réassigne plus :
\$hero->takeDamage(3) modifie le héros lui-même.



Un objet réunit un état et un comportement

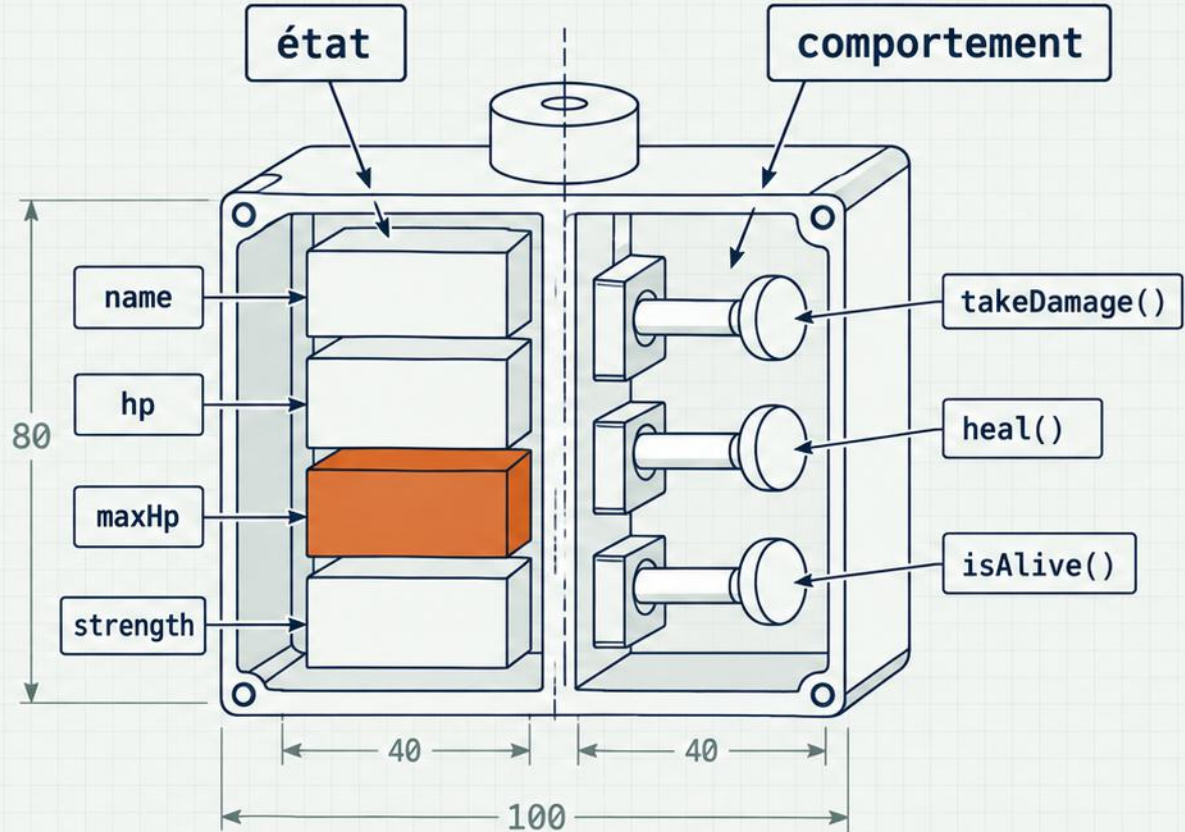
L'état :

ses propriétés,
name, hp, maxHp, strength

Le comportement :

ses méthodes,
takeDamage(), heal(), isAlive()

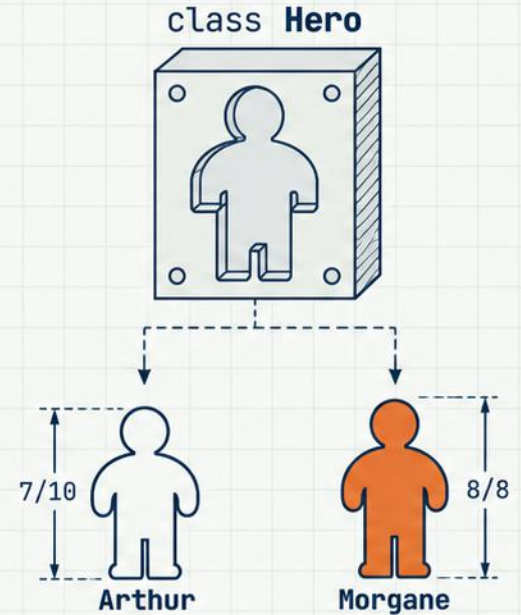
Tout le vocabulaire qui suit
sert à décrire cette phrase.



La classe est le moule, new fabrique l'instance

```
1 $arthur = new Hero('Arthur');
2 $morgane = new Hero('Morgane', maxHp: 8, strength: 3);
3
4 $arthur->takeDamage(3);
5
6 echo $arthur, PHP_EOL;
7 echo $morgane, PHP_EOL;
```

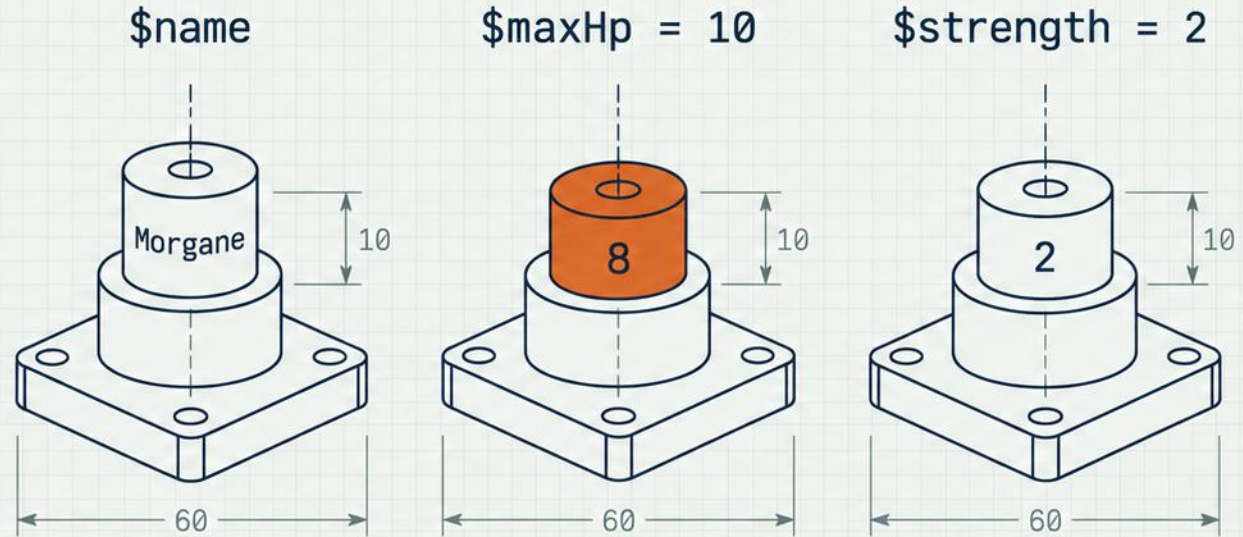
Arthur (7/10 PV)
Morgane (8/8 PV)



- 1 `class Hero { ... }` ne crée aucun **héros** : c'est le **moule**. `new Hero('Arthur')` fabrique une **instance**, avec ses propres valeurs.
- 2 Morgane n'a pas bougé : chaque héros possède son propre état.
- 3 Dans une méthode, `$this` désigne l'objet sur lequel elle a été appelée : `$arthur->takeDamage(3)` exécute `takeDamage` avec `$this` qui vaut `$arthur`.

`new Hero('Morgane', maxHp: 8, strength: 3)`

- Le constructeur attend trois paramètres : `$name` obligatoire, `$maxHp = 10` et `$strength = 2` avec une **valeur par défaut**.
- Arthur n'a reçu que son nom : 10 points de vie et 2 de force, pris par défaut.
- `maxHp: 8` est un **argument nommé** : le nom du paramètre, puis sa valeur.
- Pour ne changer que la force : `new Hero('Morgane', strength: 3)`. `maxHp` garde 10 sans qu'on écrive de 10 au milieu.



declare(strict_types=1): PHP refuse les conversions silencieuses

```
1 declare(strict_types=1);
2
3 function takeDamage(int $amount): int
4 {
5     return $amount;
6 }
7
8 takeDamage('3');
```

Fatal error: Uncaught TypeError: takeDamage(): Argument #1 (\$amount) must be of type int, string given

TEST ROUGE

Sans cette ligne, PHP accepterait '3' et le convertirait en 3 sans rien dire. Un type, c'est une vérification gratuite à chaque appel. Règle du cours : un type sur chaque propriété, chaque paramètre, chaque retour. Laravel est écrit comme ça.

Avant PHP 8, le nom était écrit trois fois

```
1 // L'ancienne façon.
2 class Hero
3 {
4     private string $name;
5
6     public function __construct(string $name)
7     {
8         $this->name = $name;
9     }
10 }
```

```
1 // Depuis PHP 8.0 : la promotion de constructeur.
2 class Hero
3 {
4     public function __construct(
5         public readonly string $name
6     ) {
7     }
8 }
```

Un mot de visibilité devant un paramètre **crée la propriété et lui donne la valeur reçue**. C'est le cas de `name` et de `strength`.

Pourquoi \$maxHp n'a pas de mot de visibilité

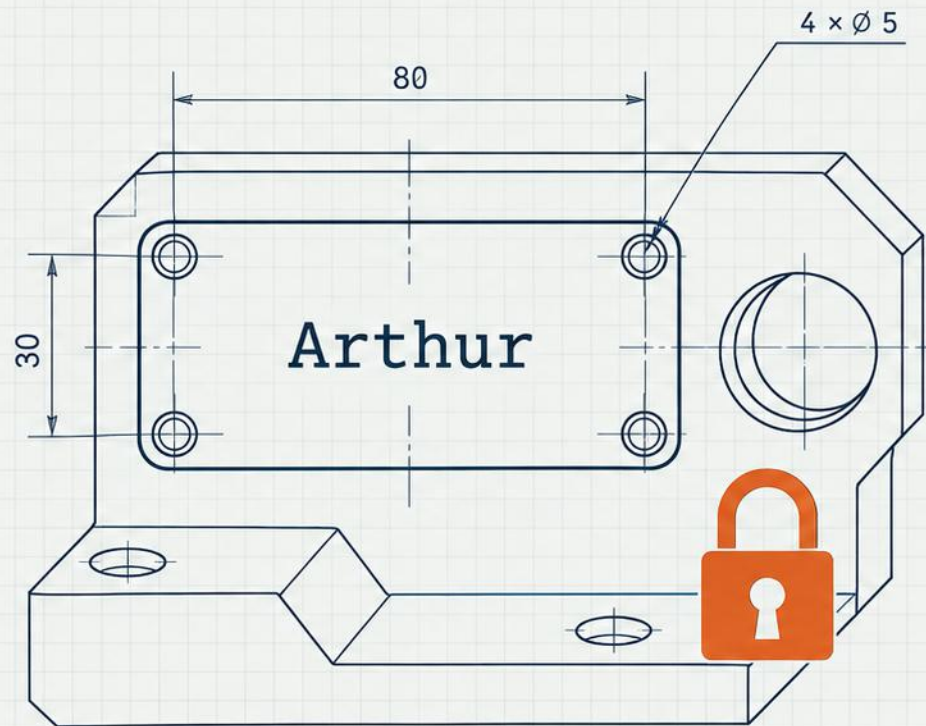
```
1  public function __construct(  
2      public readonly string $name,  
3      int $maxHp = 10,  
4      public readonly int $strength = 2,  
5  ) {  
6      $this->maxHp = $maxHp;  
7      $this->hp = $maxHp;  
8  }
```

recopié à la main

- Un paramètre est une valeur reçue le temps d'une exécution. Une propriété reste dans l'objet.
- Sans mot de visibilité, \$maxHp reste un simple paramètre : le constructeur le recopie lui-même dans la propriété.
- Le corps du constructeur sert à ce qui se calcule : hp reçoit maxHp, le héros commence avec tous ses points de vie. C'est une règle du jeu, pas un paramètre.

readonly : ce qui ne change jamais

- Un héros ne change pas de nom en cours de partie.
readonly l'écrit dans le code : la propriété reçoit sa valeur une fois, dans le constructeur, et ne peut plus être modifiée.
- **À quoi ça sert :** rendre impossible la valeur qui change par accident. Vous n'avez plus à vous demander « qui a modifié le nom ? ». Personne, c'est garanti.
- Cela rend la propriété publique sans risque :
`$arthur->name` se lit partout, ne s'écrit nulle part.
- **Réflexe :** fixé à la création et ne doit plus bouger → readonly. Le nombre de faces d'un dé, le nom d'un héros, la date d'une commande.



Qui a le droit d'y toucher ?

```
1  class Hero
2  {
3      public string $name = 'Arthur';
4      private int $hp = 10;
5
6      public function takeDamage(int $amount): void
7      {
8          $this->hp = max(0, $this->hp - $amount); // intérieur : autorisé
9      }
10 }
11
12 $arthur = new Hero();
13 echo $arthur->name, PHP_EOL; // extérieur, public : autorisé
14 $arthur->takeDamage(3);      // extérieur, méthode publique : autorisé
15 echo $arthur->hp;           // extérieur, private : refusé
```

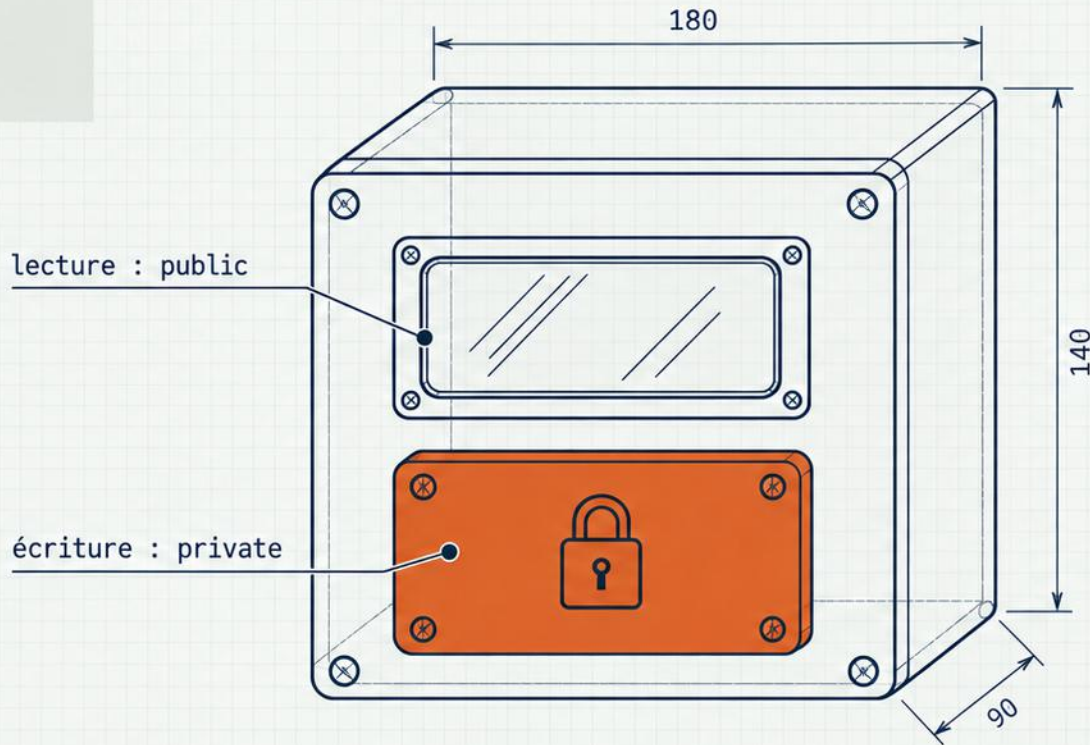
TEST ROUGE

- **L'intérieur** : le code entre les accolades de `class Hero { ... }`. **L'extérieur** : tout le reste.
- **public** : accessible de partout. **private** : depuis l'intérieur seulement. **protected** : comme private, plus les classes qui héritent • ch. 6.

public private(set) : tout le monde lit, seule la classe écrit

```
1 public private(set) int $hp;
```

- Il manque un cas : l'affichage doit lire hp pour la barre de vie, mais personne ne doit l'écrire.
- Depuis PHP 8.4, une propriété peut avoir deux visibilité : une pour la lecture, une pour l'écriture.
- `$arthur->hp` fonctionne partout.
`$arthur->hp = 9999` ne fonctionne que dans les méthodes de Hero.
- C'est la déclaration de hp et maxHp dans la classe du chapitre. Le chapitre 3 y revient.



La classe garde la main sur ses données

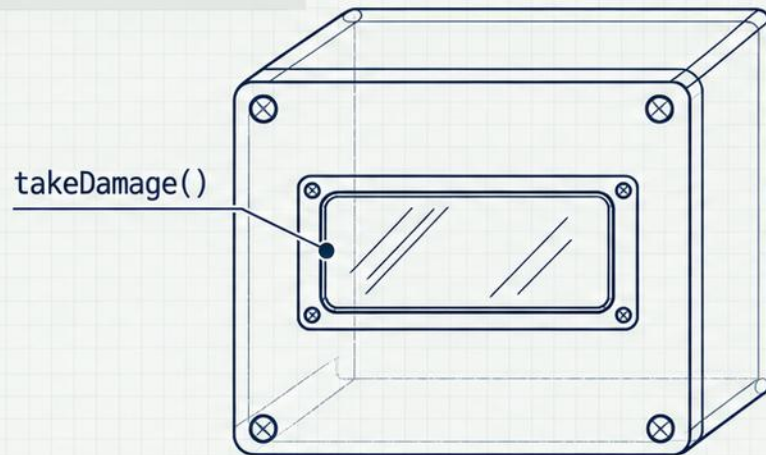
```
1 $arthur->hp = 9999;      // propriété privée en écriture  
2 $arthur->name = 'Kevin'; // propriété readonly
```

TEST ROUGE

Fatal error: Uncaught Error: Cannot modify private(set) property Hero::\$hp

Fatal error: Uncaught Error: Cannot modify readonly property Hero::\$name

- Les règles sont respectées partout : le seul chemin vers hp passe par des méthodes qui les vérifient.
- Une faute de frappe devient une erreur immédiate :
\$arthur->hpp n'existe pas, PHP le dit tout de suite.
- Vous pouvez changer l'intérieur sans casser l'extérieur.
- Le 9999 de la slide 2 est devenu impossible. C'est cela, l'encapsulation.



Pas de setHp()

- Un getter est une méthode qui se contente de lire une propriété : `getHp()`. Un setter est une méthode qui se contente de l'écrire : `setHp(int $hp)`.
- Le réflexe hérité de Java : une paire par propriété. Ne le faites pas.
- Le getter est inutile : `private(set)` ouvre déjà la lecture. Le setter est dangereux : il rouvre le trou qu'on vient de fermer.



```
1 // Mauvais : la règle « jamais sous 0 » n'est plus nulle part.  
2 public function setHp(int $hp): void { $this->hp = $hp; }  
3  
4 // Bon : l'action du jeu, avec sa règle à l'intérieur.  
5 public function takeDamage(int $amount): void { $this->hp = max(0, $this->hp - $amount); }
```

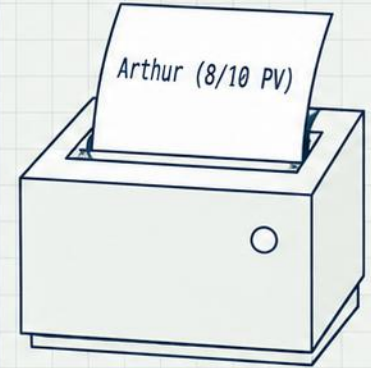
NON CONFORME

En lisant vos méthodes publiques, on doit voir les règles du jeu, pas la liste de vos propriétés.

Afficher un objet

```
1 public function __toString(): string
2 {
3     return sprintf('%s (%d/%d PV)', $this->name, $this->hp, $this->maxHp);
4 }
5
6 echo $hero, PHP_EOL;
```

Arthur (8/10 PV)



- `echo $hero` sur un objet est normalement une erreur. Déclarez `__toString()` et PHP l'appelle chaque fois que l'objet est utilisé comme du texte.
- Les noms qui commencent par deux tirets bas, `__construct`, `__toString`, sont réservés à des méthodes magiques que PHP appelle lui-même. Ces deux-là suffisent pour l'instant.

Deuxième classe du Donjon : le dé

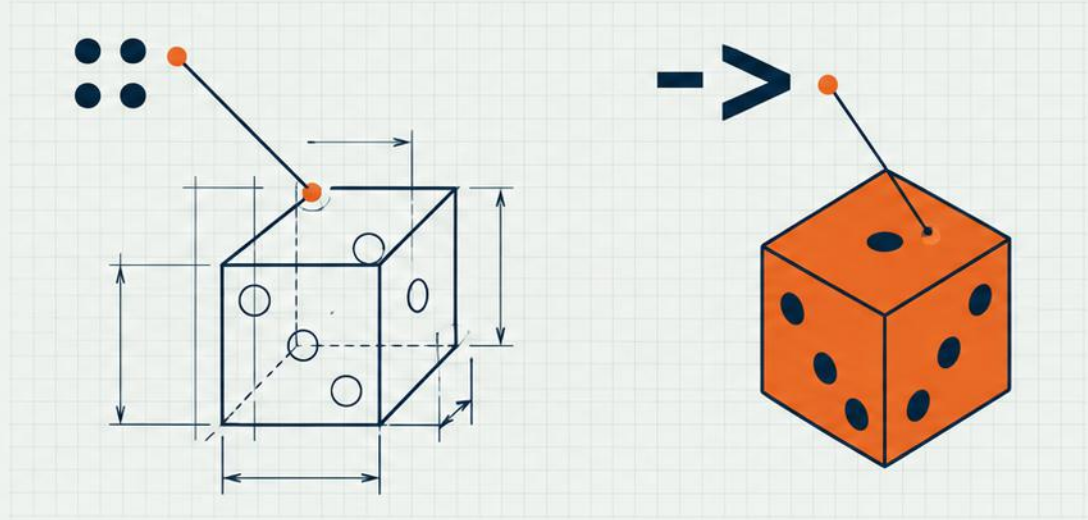
```
1  class Dice
2  {
3      public function __construct(
4          public readonly int $sides,
5      ) {
6      }
7
8      public static function d6(): self
9      {
10         return new self(6);
11     }
12
13     public function roll(): int
14     {
15         return random_int(1, $this->sides);
16     }
17 }
```

- `roll()` a besoin d'un dé précis, sans le nombre de faces impossible de tirer : **méthode d'instance**, appelée avec `->` sur un objet.
- `d6()` n'a besoin d'aucun dé existant, elle en fabrique un : **méthode statique**, appelée avec `::` sur la classe. `self` désigne `Dice`.
- Pourquoi une fabrique plutôt que `new Dice(6)` partout ? Parce que `Dice::d6()` dit ce qu'on veut obtenir.

:: parle à la classe, -> parle à un objet

```
1 $d6 = Dice::d6();  
2 echo $d6->sides, PHP_EOL;  
3 echo $d6->roll(), PHP_EOL;
```

6
4



- `Dice::d6()` : statique, adressé à la classe, fabrique un dé.
- `$d6->roll()` : instance, adressé à CE dé précis, un entier de 1 à 6 différent à chaque exécution.
- Dans Laravel, même notation : `Post::latest()` d'un côté, `$post->title` de l'autre. Le mécanisme derrière `Post::latest()` est plus subtil, mais ça se lit pareil.

Trois exercices, un fichier : hero.php

1. Deux héros indépendants :

Arthur par défaut, Morgane maxHp: 8, strength: 3. Touchez l'un, soignez l'autre, affichez les deux.

2. Une action de plus :

`attack()`: int renvoie la force du héros.

3. Deux fabriques de plus :

`Dice::d4()` et `Dice::d100()`, dix lancers chacun.

Identifiants en anglais, commentaires en français : c'est la langue de Laravel. Composer et les tests arrivent au chapitre 2.

