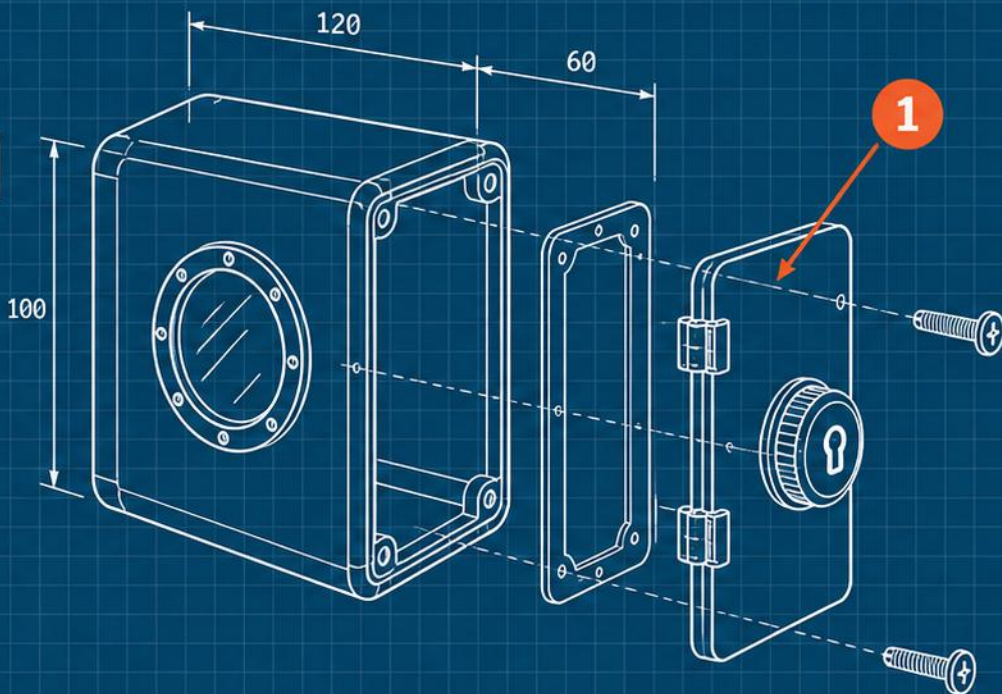


Bloc P00 · chapitre 3

L'encapsulation

Pour chaque propriété : qui lit,
qui écrit, et où vit la vérification.



Qu'affichent les deux echo ?

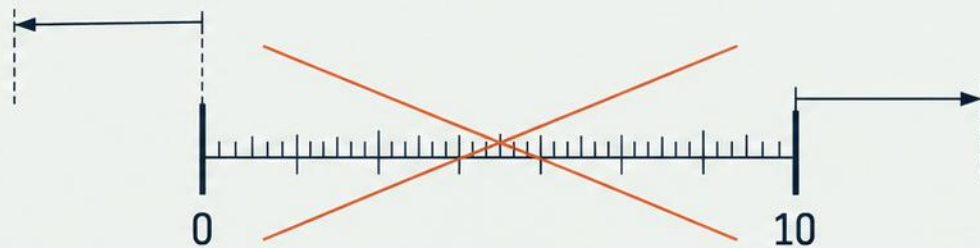
```
1  public function takeDamage(int $amount): void
2  {
3      $this->hp = max(0, $this->hp - $amount);
4  }
5
6  public function heal(int $amount): void
7  {
8      $this->hp = min($this->maxHp, $this->hp + $amount);
9  }
10
11 $arthur = new Hero('Arthur');
12 $arthur->takeDamage(-5);
13 echo $arthur, PHP_EOL;
14
15 $morgane = new Hero('Morgane');
16 $morgane->heal(-20);
17 echo $morgane, PHP_EOL;
```



Cinq minutes, en binôme. La classe **Hero** du chapitre 1, et deux appels un peu tordus.

Arthur (15/10 PV) et Morgane (-10/10 PV)

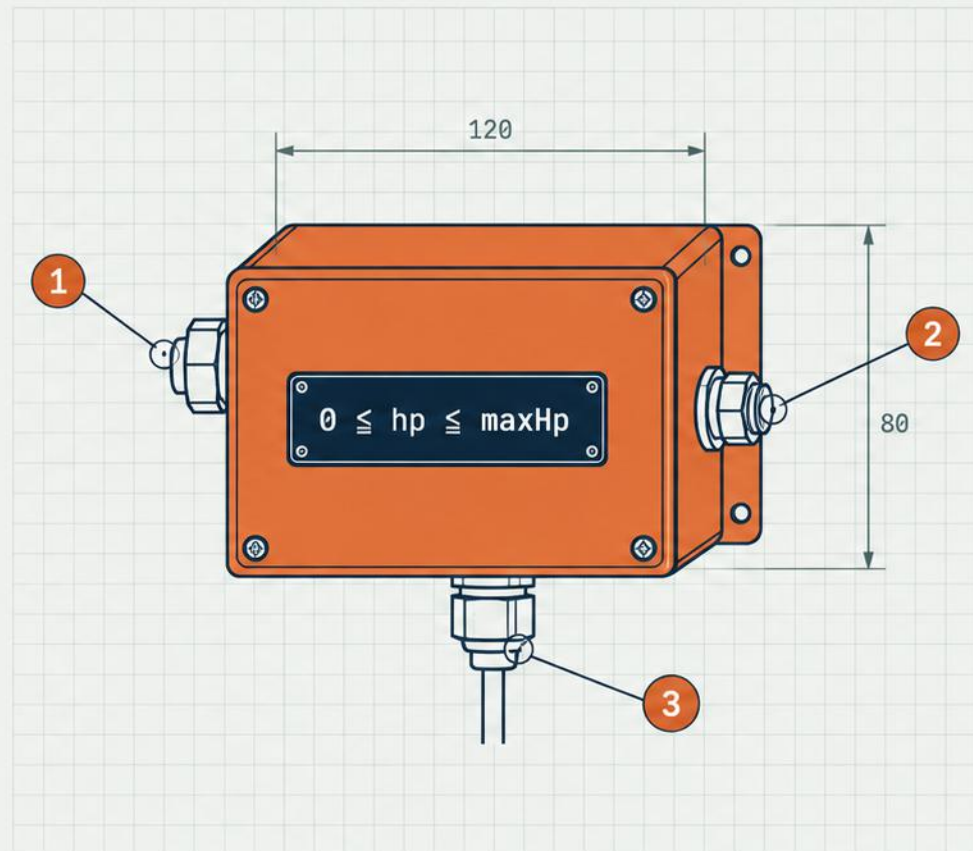
- **Un dégât négatif soigne :**
takeDamage() ne vérifie que la borne basse.
- **Un soin négatif blesse :**
heal() ne vérifie que la borne haute.
- **Fermer l'écriture avec** private(set) **ne suffit pas :** la règle « entre 0 et maxHp » est coupée en deux, et chaque moitié ignore l'autre.



NON CONFORME

hp reste entre 0 et maxHp : un invariant

- Une règle qui doit rester vraie pendant toute la vie d'un objet s'appelle un **invariant**. Le reste du programme y compte.
- **L'encapsulation** : la classe cache ses données et garantit ses règles.
- **La règle est écrite une seule fois**, dans Hero, pas dans les vingt endroits du jeu qui infligent des dégâts.
- **Un bug se cherche à un seul endroit** : si hp vaut 9999, l'erreur est dans Hero.
- **Les méthodes publiques suffisent à comprendre la classe** : `takeDamage()`, `heal()`, `isAlive()`.

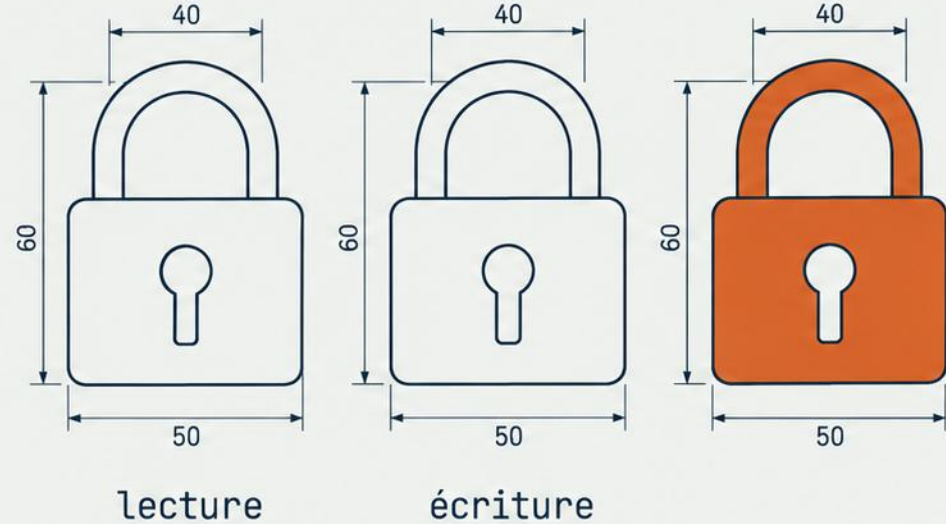


Quatre cas pour une propriété

Déclaration	Qui lit	Qui écrit	Quand
<code>private int \$turn</code>	la classe	la classe	un détail interne
<code>public readonly string \$name</code>	tout le monde	personne, après le constructeur	fixée à la création
<code>public private(set) int \$hp</code>	tout le monde	la classe	change par les actions de la classe
<code>public int \$hp</code>	tout le monde	tout le monde	jamais, dans ce cours

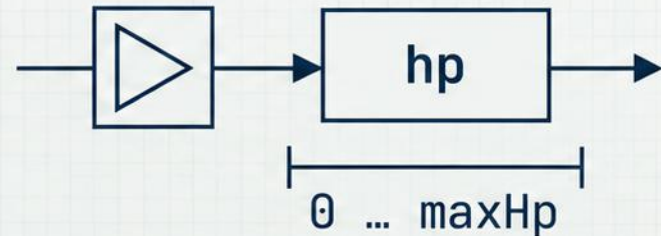
Écrivez private. Ouvrez la lecture si besoin. N'ouvrez jamais l'écriture.

1. Écrivez private. Tout autre choix demande une raison. Un compteur de tours dans Dungeon reste `private int $turn = 0`.
2. Ouvrez la lecture quand quelqu'un en a besoin. L'affichage doit lire `hp : public private(set)`. Le nom ne change jamais : `public readonly`.
3. N'ouvrez jamais l'écriture. Modifier un héros, c'est une action du jeu, et le nom de cette action contient un verbe : `takeDamage()`, `heal()`, `equip()`.
4. `protected` attend le chapitre 6 : il sert quand une classe hérite d'une autre.



La règle vit sur la propriété

```
1 public private(set) int $hp = 0 {  
2     set => max(0, min($this->maxHp, $value));  
3 }
```



- Depuis PHP 8.4, on peut attacher du code à la propriété elle-même. C'est un hook : du code exécuté à chaque lecture ou à chaque écriture. Il y en a deux, set et get.
- \$value est fournie par PHP : la valeur qu'on a tenté d'écrire. L'expression après => donne la valeur réellement stockée.
- Avec maxHp à 10, après 50 dégâts, la valeur proposée est -40. min(10, -40) donne -40, puis max(0, -40) donne 0.

Toutes les écritures passent par le hook

```
1  public function takeDamage(int $amount): void
2  {
3      $this->hp -= $amount;
4  }
5
6  public function heal(int $amount): void
7  {
8      $this->hp += $amount;
9  }
10
11 $arthur = new Hero('Arthur');
12 $arthur->takeDamage(50);
13 echo $arthur, PHP_EOL;
14 $arthur->heal(999);
15 echo $arthur, PHP_EOL;
```

Arthur (0/10 PV)
Arthur (10/10 PV)

VÉRIFIÉ

Les deux bornes sont vérifiées dans le hook de hp. Une potion empoisonnée pourra écrire `$this->hp -= 3` sans rien vérifier.

setHp(), refusé au chapitre 1, ouvrirait l'écriture à tout le programme

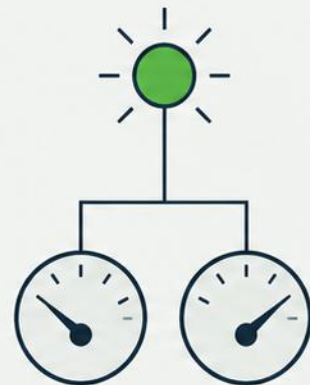
```
1 public private(set) int $hp = 0 {  
2     set {  
3         $this->hp = max(0, min($this->maxHp, $value));  
4     }  
5 }
```



- Avec les accolades, vous écrivez vous-même `$this->hp = ...` : le hook remplace l'écriture, il ne s'y ajoute pas. Cette forme sert quand il faut plusieurs instructions, par exemple un `throw`.
- Ici, `private(set)` réserve l'écriture à Hero, et le hook vérifie la valeur que Hero écrit.
- Seule exception : la valeur par défaut de la déclaration, `= 0`, ne passe pas par le hook.

Une valeur calculée, jamais stockée

```
1 public bool $isFullHealth {  
2     get => $this->hp === $this->maxHp;  
3 }  
4  
5 $arthur = new Hero('Arthur');  
6 var_dump($arthur->isFullHealth);  
7 $arthur->takeDamage(1);  
8 var_dump($arthur->isFullHealth);  
9 $arthur->isFullHealth = true;
```



```
bool(true)
```

```
bool(false)
```

```
Fatal error: Uncaught Error: Property Hero::$isFullHealth is read-only
```

- Un **booléen** est une valeur vraie ou fausse. Quand vous lisez `$arthur->isFullHealth`, PHP exécute le code du `get`.
- La propriété est **virtuelle** : rien n'est stocké, la valeur est calculée à chaque lecture.
- Sans `set`, personne ne peut l'écrire.

Un état se lit, une action se fait

- **Une propriété** pour ce qui se lit comme un état :
hp, name, isFullHealth.
- **Une méthode** pour une action, dont le nom contient un verbe et qui change quelque chose :
takeDamage(), heal(), roll().
- **Une méthode** pour un calcul qui a besoin d'un paramètre : `distanceTo(Point $other)`.
- Entre `isAlive()` et `isAlive`, les deux écritures se valent. Le Donjon garde la méthode. Ce qui compte : aucune des deux ne s'écrit de l'extérieur.



Un objet impossible ne se construit pas

```
1 class Dice
2 {
3     public function __construct(
4         public readonly int $sides,
5     ) {
6         if ($sides < 2) {
7             throw new \InvalidArgumentException("Un dé a au moins 2 faces, $sides reçu.");
8         }
9     }
10 }
11
12 $d0 = new Dice(0);
```

Fatal error: Uncaught InvalidArgumentException: Un dé a au moins 2 faces, 0 reçu.

TEST ROUGE

- Une propriété `readonly` ne peut pas avoir de hook : sa règle se vérifie dans le constructeur.
- Une **exception** est un objet qui signale un problème et interrompt le code en cours. `throw` la lance : le constructeur s'arrête, `new` ne renvoie aucun dé.
- `InvalidArgumentException` est la classe fournie par PHP pour ce cas : l'argument reçu n'est pas acceptable. Chapitre 6 pour la suite.

Un nom vide, un maximum nul : refusés

```
1  if (trim($name) === '') {  
2      throw new \InvalidArgumentException('Un héros a un nom.');
```



```
3  }  
4  
5  if ($maxHp < 1) {  
6      throw new \InvalidArgumentException("maxHp doit valoir au moins 1, $maxHp reçu.");  
7  }
```

Fatal error: Uncaught InvalidArgumentException: Un héros a un nom.

Fatal error: Uncaught InvalidArgumentException: maxHp doit valoir au moins 1, 0 reçu.

Dans le constructeur, avant les deux affectations. trim() enlève les espaces aux deux bouts, pour qu'un nom fait d'espaces soit refusé aussi. Chaque essai se lance seul :
new Hero(' '), puis new Hero('Arthur', 0).

Trois règles, écrites une seule fois

```
1  <?php
2
3  declare(strict_types=1);
4
5  class Hero
6  {
7      public private(set) int $maxHp;
8
9      public private(set) int $hp = 0 {
10         set => max(0, min($this->maxHp, $value));
11     }
12
13     public bool $isFullHealth {
14         get => $this->hp === $this->maxHp;
15     }
16
17     public function __construct(
18         public readonly string $name,
19         int $maxHp = 10,
20         public readonly int $strength = 2,
21     ) {
22         if (trim($name) === '') {
23             throw new \InvalidArgumentException('Un héros a un nom.');
```

1 — règle 1

2 — règle 2

3 — règle 3

```
24         }
25
26         if ($maxHp < 1) {
27             throw new \InvalidArgumentException("maxHp doit valoir au moins 1, $maxHp reçu.");
28         }
29
30         $this->maxHp = $maxHp;
31         $this->hp = $maxHp;
32     }
}
```

Les bornes de hp, le nom non vide, le maximum positif. Quinze lignes de plus qu'au chapitre 1.

Les méthodes n'ont plus rien à vérifier

```
33 public function takeDamage(int $amount): void
34 {
35     $this->hp -= $amount;
36 }
37
38 public function heal(int $amount): void
39 {
40     $this->hp += $amount;
41 }
42
43 public function isAlive(): bool
44 {
45     return $this->hp > 0;
46 }
47
48 public function __toString(): string
49 {
50     return sprintf('%s (%d/%d PV)', $this->name, $this->hp, $this->maxHp);
51 }
52 }
```

actions

```
Arthur (0/10 PV) false
Arthur (10/10 PV) true
```

Suite du même fichier. Après 50 dégâts puis 50 soins, isFullHealth passe de false à true sans qu'aucune méthode ne s'en occupe.

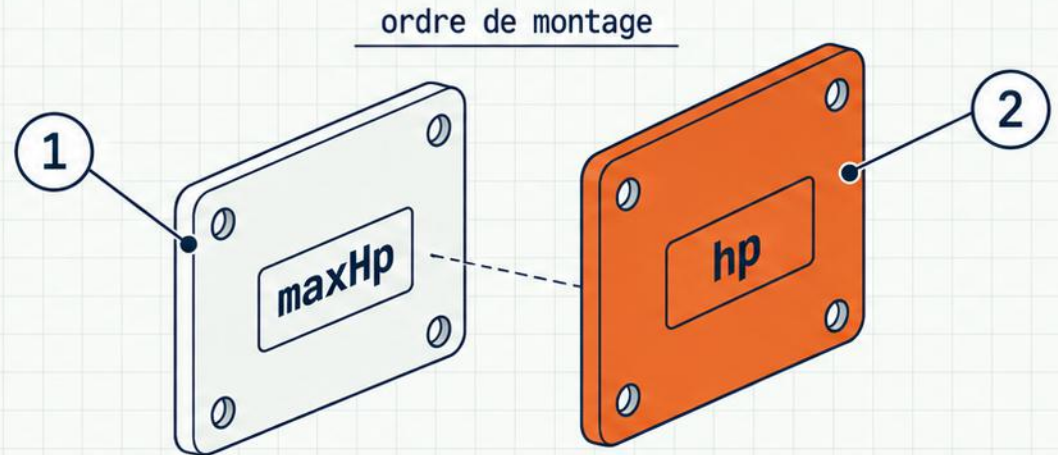
maxHp d'abord, hp ensuite

```
1 $this->maxHp = $maxHp;  
2 $this->hp = $maxHp;
```

Fatal error: Uncaught Error: Typed property Hero::\$maxHp must not be accessed before initialization

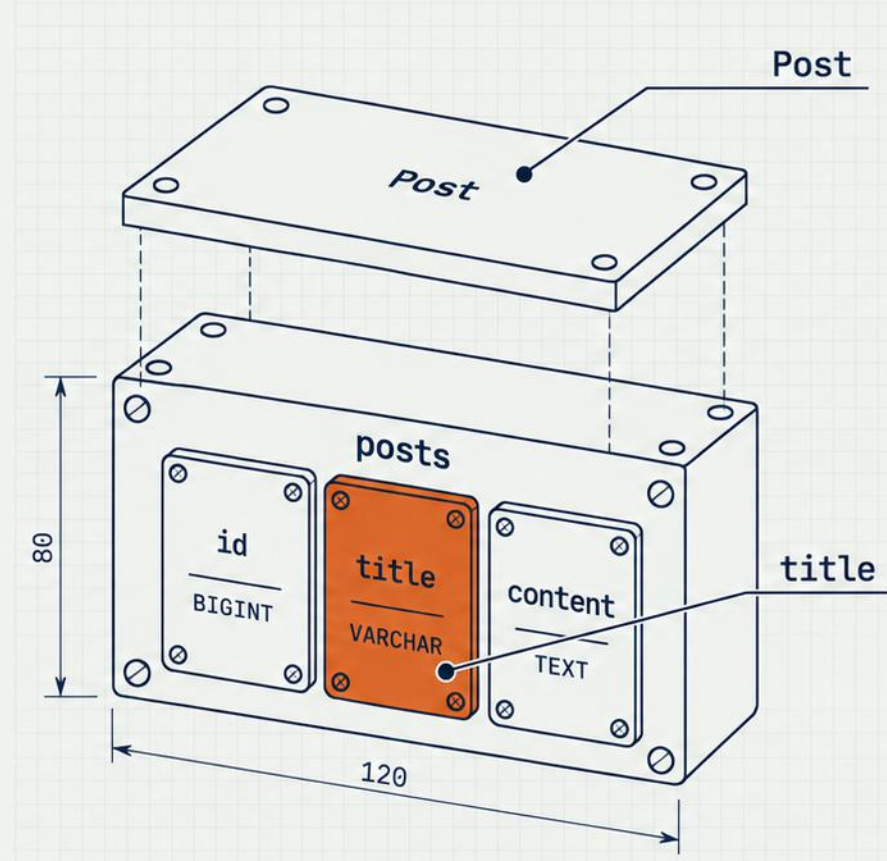
TEST ROUGE

- Le hook de hp lit maxHp. Si hp reçoit sa valeur en premier, maxHp n'en a pas encore, et PHP s'arrête.
- maxHp reste private(set) et non readonly : un objet pourra un jour augmenter le maximum. Une propriété readonly l'interdirait.



\$post->title : le modèle décide

- Vous écrirez `$post->title` pour lire un titre et `$post->title = 'Le donjon'` pour le changer, avant d'enregistrer avec `save()`.
- Ces propriétés n'existent pas dans le fichier `Post.php` : le modèle les fabrique à partir des colonnes de la table, avec les méthodes magiques `__get()` et `__set()`, cousines du `__toString()` du chapitre 1.
- C'est le modèle qui décide ce qui se lit, ce qui s'écrit et ce qui se convertit au passage. Le chapitre 10 le montre sur du vrai code.



Kata : encapsulation

```
composer test -- --group=encapsulation
```

- 1. Le hook sur hp**, puis `takeDamage()` et `heal()` en une ligne chacune.
- 2. `isFullHealth`** : remplacez « À implémenter » par le calcul.
- 3. Valider à l'entrée** : `new Dice(1)`, `new Hero(' ')`, `new Hero('Arthur', 0)` lèvent une `InvalidArgumentException`.
- 4. Refuser les montants négatifs** dans `takeDamage()` et `heal()`, puis écrivez le test qui le vérifie, terminé par `->group('encapsulation')`.
- 5. Poussez** : la tâche `encapsulation` doit être verte, à côté de niveau-1.

Puis, sur papier : les trois propriétés de `Room`, avec leur visibilité de lecture et d'écriture.

