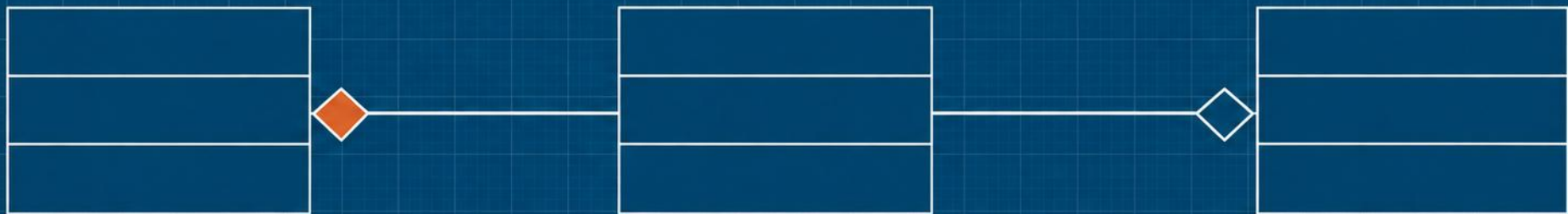


Bloc P00 · chapitre 5

Relations entre objets : posséder, partager, pointer

Un losange plein, un losange vide, un trait : chacun a sa traduction en PHP.



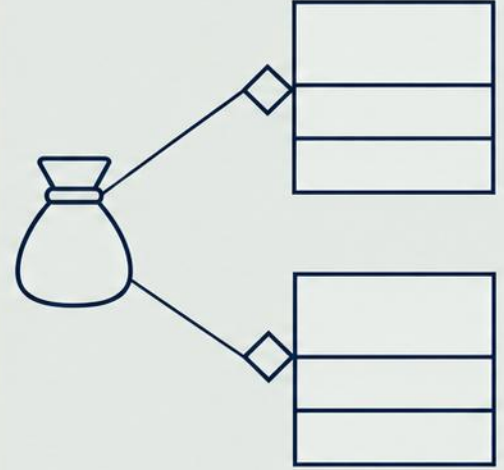
Un sac, deux héros : qu'affiche la dernière ligne ?

```
1 class Inventory
2 {
3     /** @var string[] */
4     public array $items = [];
5 }
6
7 class Hero
8 {
9     public function __construct(
10         public readonly string $name,
11         public Inventory $inventory,
12     ) {}
13 }
```

Cinq minutes, en binôme. Ce code tourne. La suite du fichier est sur la slide suivante.

Suite du même fichier : le sac passe deux fois

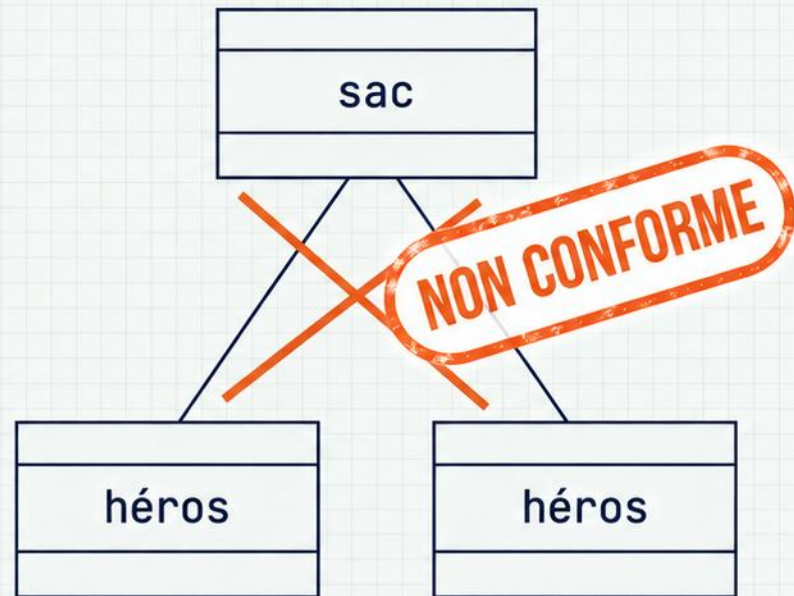
```
14 $sac = new Inventory();  
15 $arthur = new Hero('Arthur', $sac);  
16 $morgane = new Hero('Morgane', $sac);  
17  
18 $arthur->inventory->items[] = 'Épée courte';  
19  
20 print_r($morgane->inventory->items);
```



- Qu'affiche `print_r($morgane->inventory->items)` ?

Morgane a l'épée d'Arthur : un seul sac pour deux héros

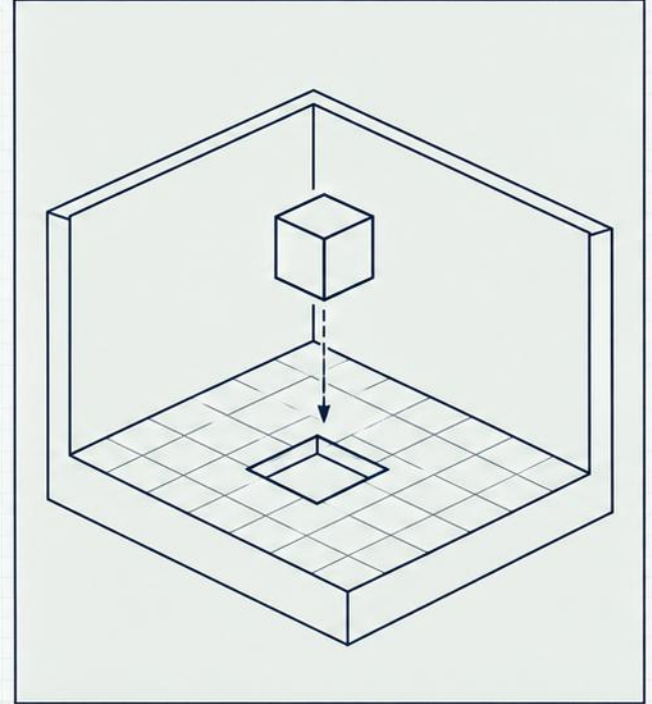
- Le sac est reçu en paramètre au lieu d'être créé dans le constructeur. Rien n'empêche de donner le même sac à deux héros.
- `$sac` passé deux fois ne fabrique pas deux sacs : c'est le même objet.
- `public array $items` laisse écrire dans le sac sans passer par une méthode, donc sans contrôle du poids.



Le sac d'un héros doit naître dans le constructeur de ce héros.

Une propriété peut contenir un objet, pas seulement un int ou une string

```
1  class Room
2  {
3      /** L'objet posé au sol. Au plus un, d'où le point d'interrogation. */
4      public private(set) ?Item $loot = null;
5
6      public function __construct(
7          public readonly string $name,
8      ) {}
9
10     /** Pose un objet au sol. */
11     public function drop(Item $item): void
12     {
13         $this->loot = $item;
14     }
15 }
```

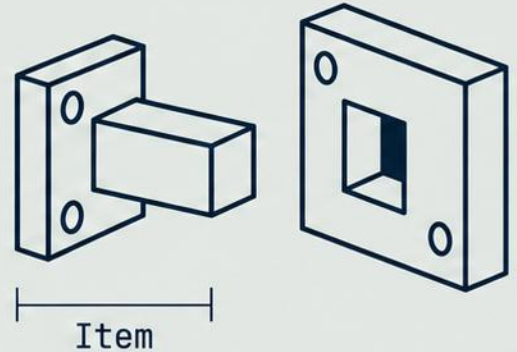


?Item \$loot : cette propriété contient un objet Item, ou null. Le paramètre de drop() porte le même type, sans le point d'interrogation.

Une chaîne de caractères ne passe pas : le programme s'arrête à la ligne fautive

```
1 $tresor = new Room('Trésor');  
2 $tresor->drop('Épée courte');
```

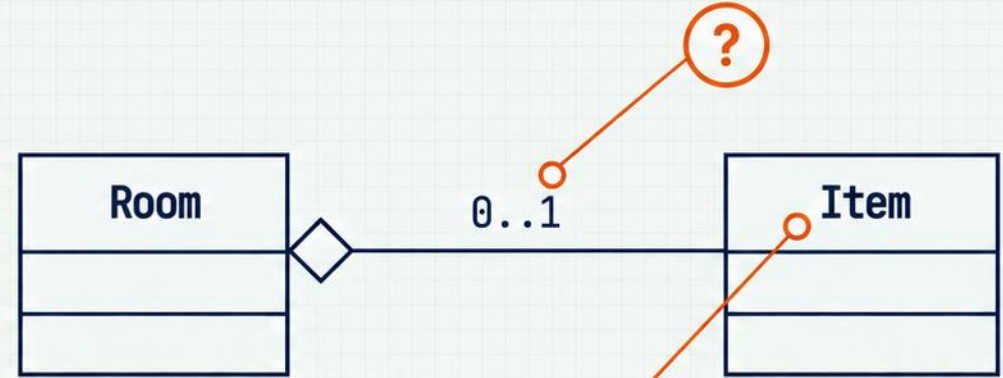
Fatal error: Uncaught TypeError: Room::drop(): Argument #1 (\$item) must be of type Item, string given



Le bug ne va pas plus loin. Le type du paramètre protège la propriété.

Une propriété typée par une classe est une relation

- Le **type** dit avec quelle classe : Item.
- Le **point d'interrogation** dit combien : zéro ou un, la multiplicité 0..1 du chapitre 4.
- ?Item \$loot est le trait entre Room et Item, le losange vide, écrit en PHP.

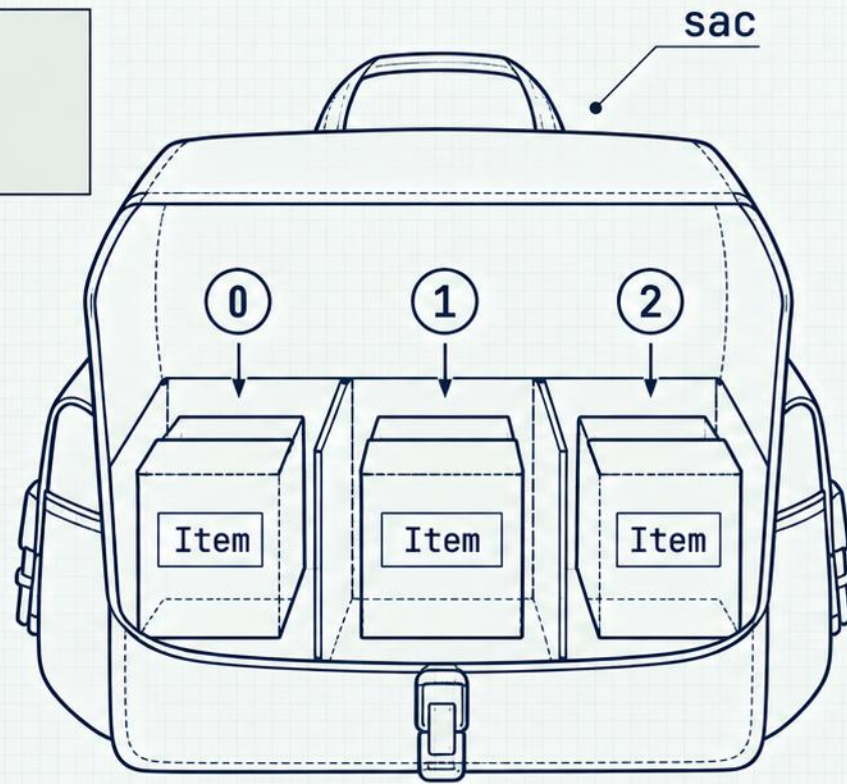


```
public private(set) ?Item $loot = null;
```

Quand la multiplicité passe à 0..*, la propriété devient un tableau

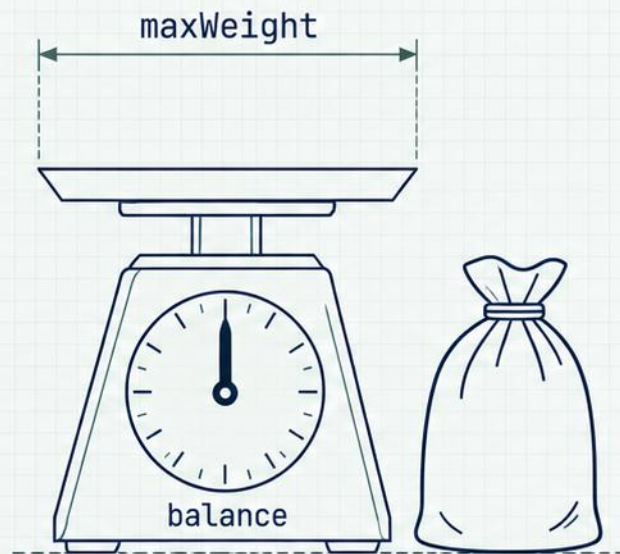
```
1  /** @var Item[] Les objets transportés. */  
2  private array $items = [];
```

- Le type array ne dit pas ce que le tableau contient. Le commentaire @var Item[] le dit à l'éditeur, qui propose alors les méthodes d'Item dès que vous tapez \$item->.
- PHP ne lit pas ce commentaire. Ce qui protège le tableau, c'est le type du paramètre d'add().
- Écrivez-le à chaque tableau d'objets.



Le contrôle du poids vit dans add(), à un seul endroit

```
1  /** Ajoute un objet, sauf si le sac déborde. */
2  public function add(Item $item): bool
3  {
4      if ($this->totalWeight() + $item->weight > $this->maxWeight) {
5          return false;
6      }
7
8      $this->items[] = $item;
9
10     return true;
11 }
```

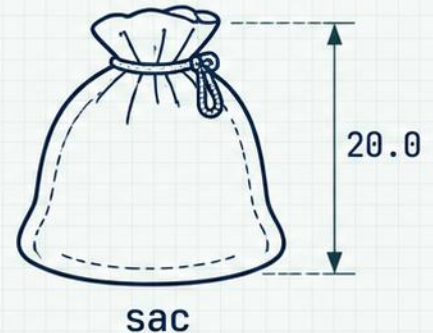
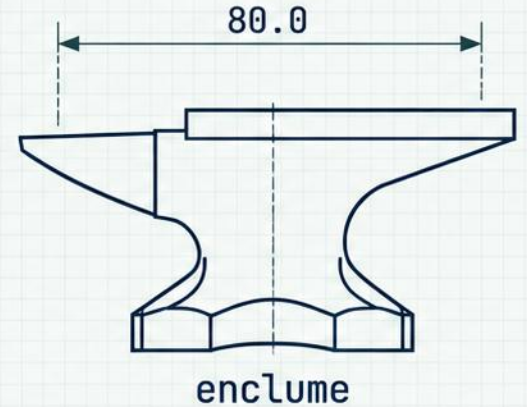


- `$this->items[] = $item;` ajoute l'objet à la fin du tableau. Aucune écriture dans `$items` ne vient de l'extérieur : la propriété est *private*.
- `add()` renvoie `false` quand le sac déborde, et n'ajoute rien. Au chapitre 6, une exception signalera l'échec à la place.

L'enclume est refusée : 2,5 plus 80 dépasse les 20 kg du sac

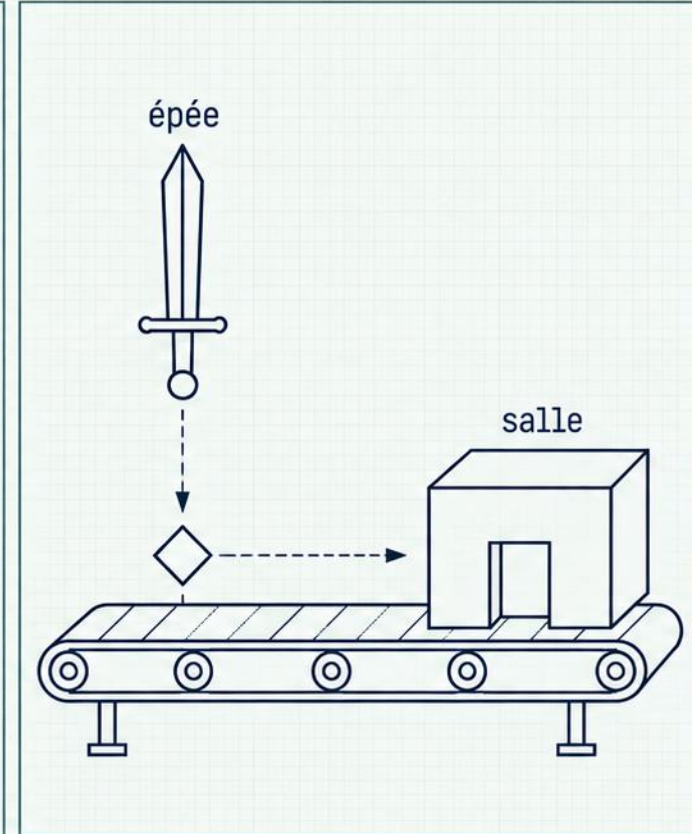
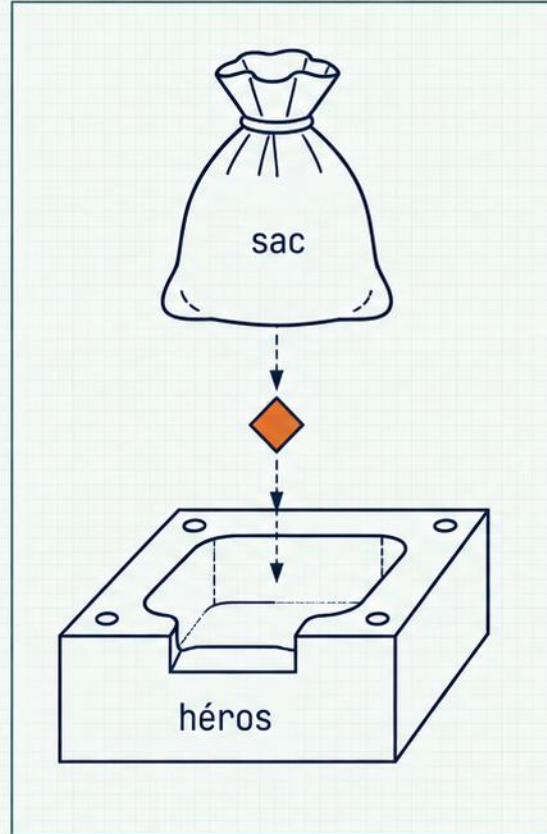
```
1 $arthur = new Hero('Arthur');  
2 $arthur->inventory->add(new Item('Épée courte', 2.0));  
3 $arthur->inventory->add(new Item('Potion', 0.5));  
4  
5 echo 'Enclume acceptée ? ', var_export($arthur->inventory->add(new Item('Enclume', 80.0), true), "\n";  
6 echo $arthur->inventory->count(), ' objets, ', $arthur->inventory->totalWeight(), " kg\n";
```

Enclume acceptée ? false
2 objets, 2.5 kg



Composition ou agrégation : le modèle du jeu tranche, new est l'indice

- La **composition** dit que la partie est fabriquée par le tout et n'appartient qu'à lui.
- L'**agrégation** dit que la partie vient d'ailleurs et lui survit.
- C'est le modèle du jeu qui tranche, pas le code. Une fois la décision prise, un indice la donne : **qui appelle new ?**



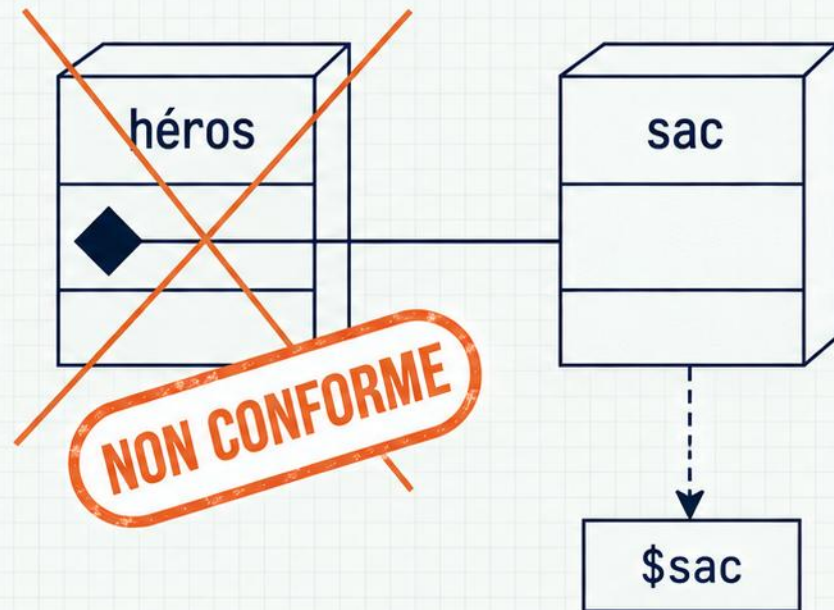
new Inventory() dans le constructeur : le losange plein

```
1  class Hero
2  {
3
4      /** Le sac : créé ici, et nulle part ailleurs. Lisible de partout, jamais remplacé. */
5      public readonly Inventory $inventory;
6
7      public function __construct(
8          public readonly string $name,
9          int $maxHp = 10,
10         public readonly int $strength = 2,
11     ) {
12         $this->maxHp = $maxHp;
13         $this->hp = $maxHp;
14         $this->inventory = new Inventory();
15     }
16 }
```

Trois conséquences. Personne ne peut donner à un héros un sac déjà rempli. Deux héros ne peuvent pas recevoir le même sac. Le sac n'a aucune raison d'exister ailleurs que chez son héros.

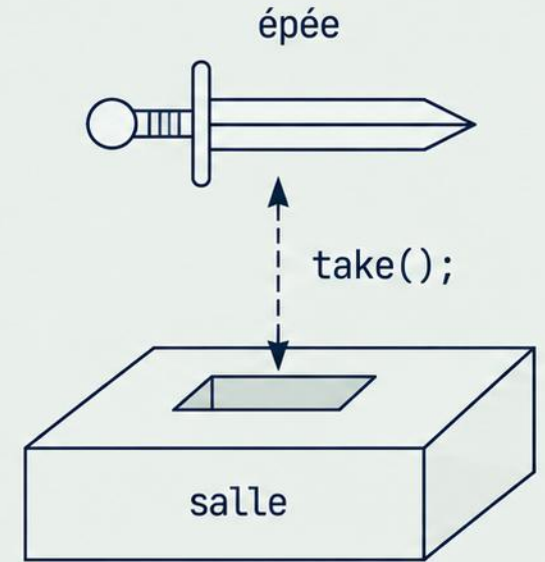
« La partie meurt avec le tout » est une règle du modèle, pas une garantie du langage

- PHP libère un objet quand plus aucune variable ne le désigne.
- Si vous écrivez `$sac = $arthur->inventory;` puis que vous oubliez Arthur, le sac reste accessible par `$sac`.
- Le losange plein dit une intention. C'est à votre code de la tenir.



Aucun new Item dans Room : le losange vide

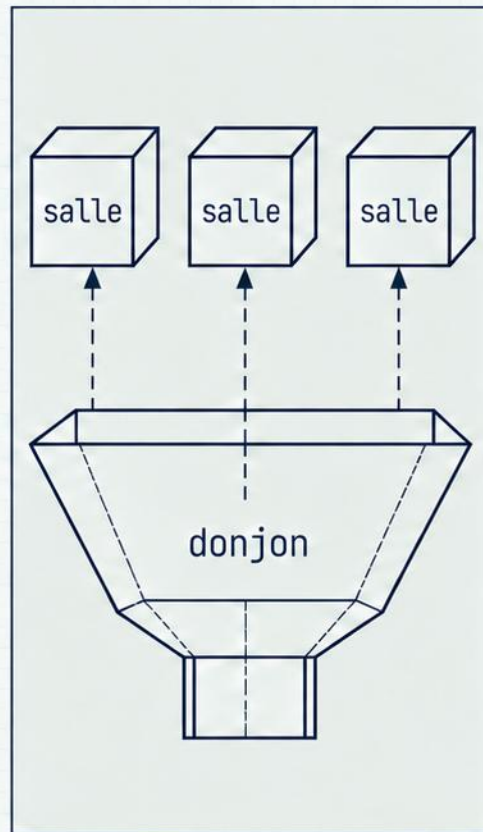
```
1  /** Pose un objet au sol. Vue à la section 1. */
2  public function drop(Item $item): void
3  {
4      $this->loot = $item;
5  }
6
7  /** Retire l'objet du sol et le rend. */
8  public function take(): ?Item
9  {
10     $item = $this->loot;
11     $this->loot = null;
12
13     return $item;
14 }
```



- La salle reçoit un objet fabriqué ailleurs, elle le garde un moment, elle le rend.

Une liste de noms, pas une liste de salles : composition

```
1  class Dungeon
2  {
3      /** @var Room[] Les salles, fabriquées ici. Lisibles de partout, modifiées seulement ici. */
4      public private(set) array $rooms = [];
5
6      /** @param string[] $names */
7      public function __construct(array $names)
8      {
9          foreach ($names as $name) {
10             $this->rooms[] = new Room($name);
11          }
12      }
13
14      public function room(int $index): Room
15      {
16          return $this->rooms[$index];
17      }
18  }
```

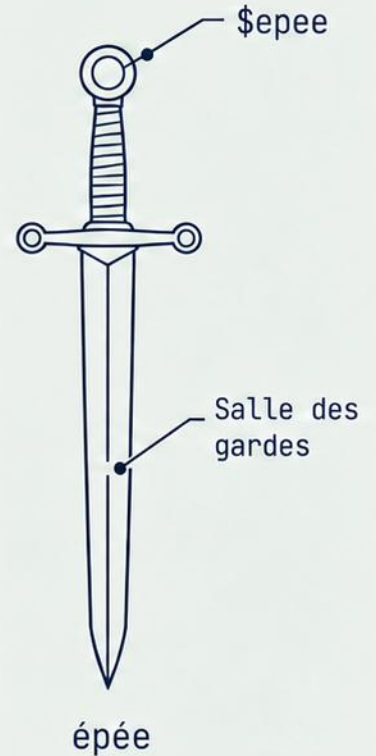


- Personne ne peut lui passer une salle déjà peuplée. C'est une composition, comme le sac.

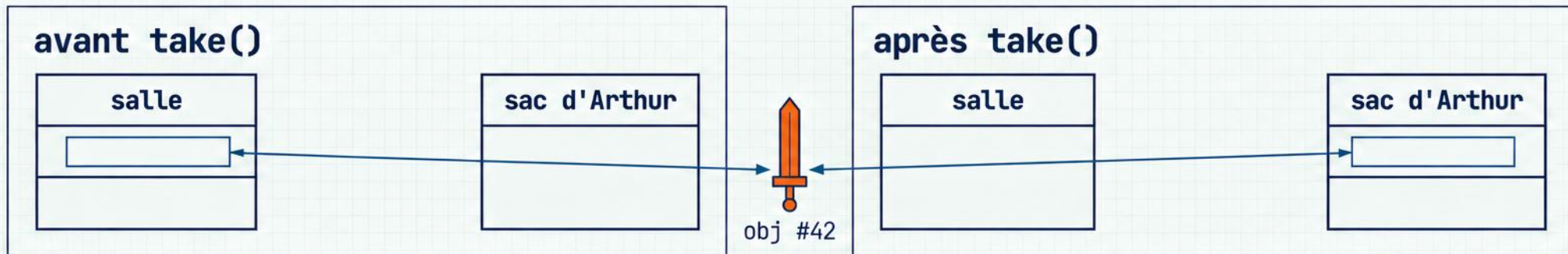
La salle détient votre épée, pas une copie

```
1 $dungeon = new Dungeon(['Entrée', 'Salle des gardes', 'Trésor']);
2
3 $epee = new Item('Épée courte', 2.0);
4 $dungeon->room(1)->drop($epee);
5
6 foreach ($dungeon->rooms as $room) {
7     echo $room->describe(), "\n";
8 }
9
10 echo 'même objet que $epee ? ', var_export($dungeon->room(1)->loot === $epee, true), "\n";
```

Entrée : rien au sol
Salle des gardes : Épée courte
Trésor : rien au sol
même objet que \$epee ? true



Une propriété contient une référence : de quoi retrouver l'objet, pas l'objet lui-même

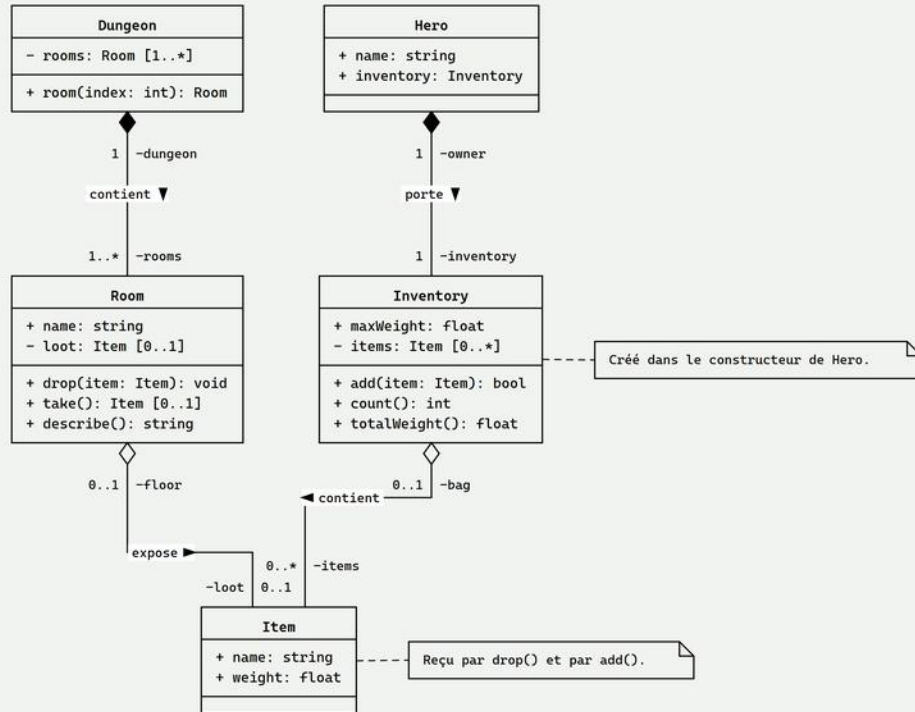


```
1 $arthur = new Hero('Arthur');
2 $ramasse = $dungeon->room(1)->take();
3
4 if ($ramasse !== null) {
5     $arthur->inventory->add($ramasse);
6 }
7
8 echo 'toujours la même épée ? ', var_export($ramasse === $epee, true), "\n";
```

toujours la même épée ? true

take() déplace la référence, pas l'objet. Une seule épée en mémoire, deux propriétaires successifs.

Un héros est fait d'un sac, une salle a une épée

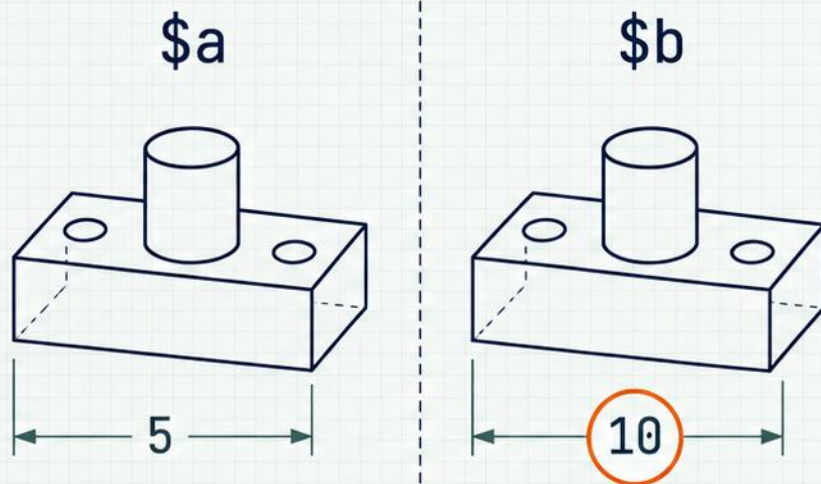


- **Composition** : le héros fabrique le sac, et le sac n'a pas de sens sans lui. Losange plein.
- **Agrégation** : l'épée existait avant la salle. Losange vide.
- Chaque relation se lit deux fois : sur le trait, et comme propriété dans la boîte qui la porte. `rooms`, `loot` et `items` sont notées – parce que seule leur classe les écrit.

Avec les types de base, PHP copie la valeur

```
1 $a = 5;  
2 $b = $a;  
3 $b = 10;  
4  
5 echo $a, "\n";
```

5

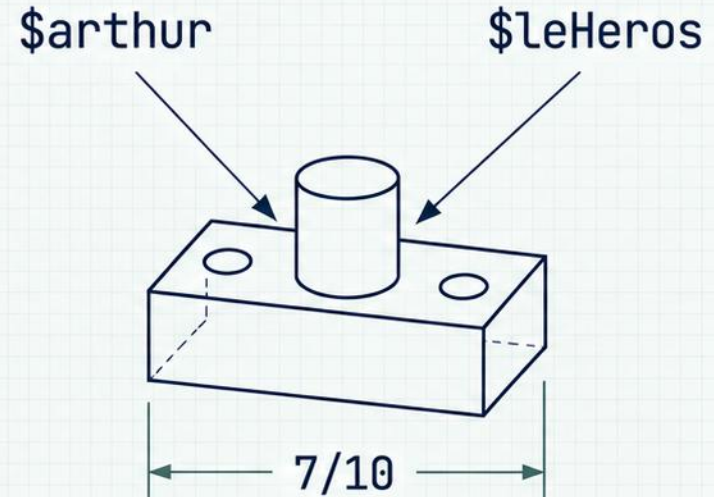


`$b` a reçu une copie. Changer `$b` ne touche pas `$a`.

Avec les objets, non : \$leHeros = \$arthur recopie la référence, pas l'objet

```
1 $arthur = new Hero('Arthur');  
2 $leHeros = $arthur;  
3  
4 $leHeros->takeDamage(3);  
5  
6 echo $arthur, "\n";  
7 echo $leHeros, "\n";
```

```
Arthur (7/10 PV)  
Arthur (7/10 PV)
```

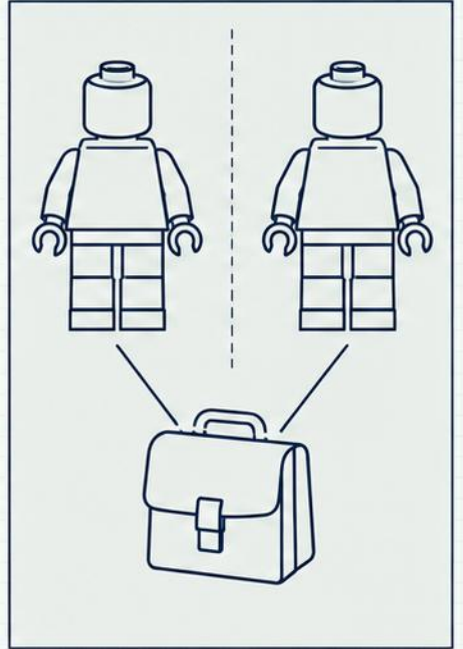


Vous avez blessé Arthur en passant par \$leHeros : il n'y a jamais eu qu'un seul héros. C'est l'explication du code d'ouverture.

clone fabrique un second héros, et recopie la référence au sac

```
1 $arthur = new Hero('Arthur');  
2 $arthur->inventory->add(new Item('Épée courte', 2.0));  
3  
4 $jumeau = clone $arthur;  
5 $jumeau->inventory->add(new Item('Potion', 0.5));  
6  
7 echo 'même sac ? ', var_export($jumeau->inventory === $arthur->inventory, true), "\n";  
8 echo 'Arthur : ', $arthur->inventory->count(), ' / jumeau : ', $jumeau->inventory->count(), "\n";
```

```
même sac ? true  
Arthur : 2 / jumeau : 2
```

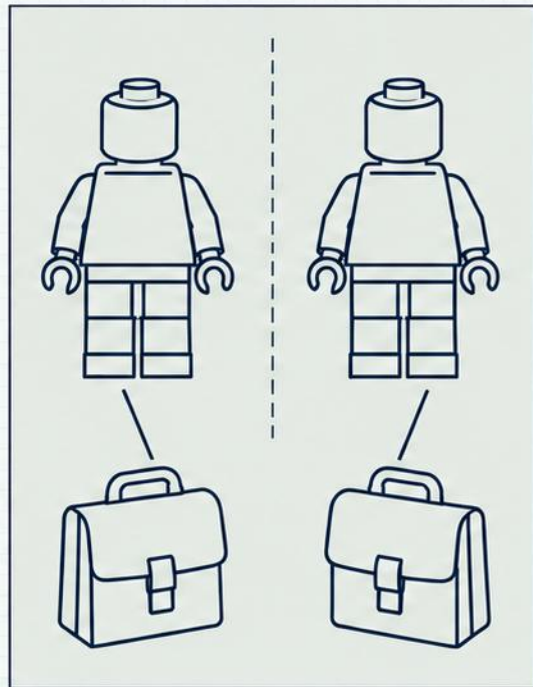


La potion du jumeau a atterri chez Arthur : clone recopie les propriétés une à une, et inventory contient une référence.

__clone() : PHP l'appelle sur la copie, juste après l'avoir fabriquée

```
1  class Hero
2  {
3      // ... tout le reste ne change pas.
4
5      /** Appelée sur la copie : le sac est recopié, pas partagé. */
6      public function __clone(): void
7      {
8          $this->inventory = clone $this->inventory;
9      }
10 }
```

même sac ? false
Arthur : 1 / jumeau : 2

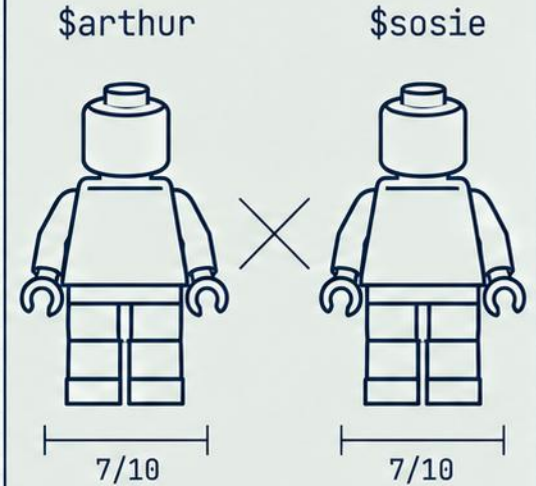


Cette copie s'appelle une **copie profonde** : chaque héros a son sac.
Les objets du sac restent partagés, et ça ne gêne pas : un **Item** est entièrement **readonly**.

== compare le contenu, === compare l'identité

```
1 $arthur = new Hero('Arthur');
2 $leHeros = $arthur;
3 $leHeros->takeDamage(3);
4
5 $sosie = new Hero('Arthur');
6 $sosie->takeDamage(3);
7
8 echo 'arthur === leHeros : ', var_export($arthur === $leHeros, true), "\n";
9 echo 'arthur === sosie   : ', var_export($arthur === $sosie, true), "\n";
10 echo 'arthur == sosie    : ', var_export($arthur == $sosie, true), "\n";
```

```
arthur === leHeros : true
arthur === sosie   : false
arthur == sosie    : true
```



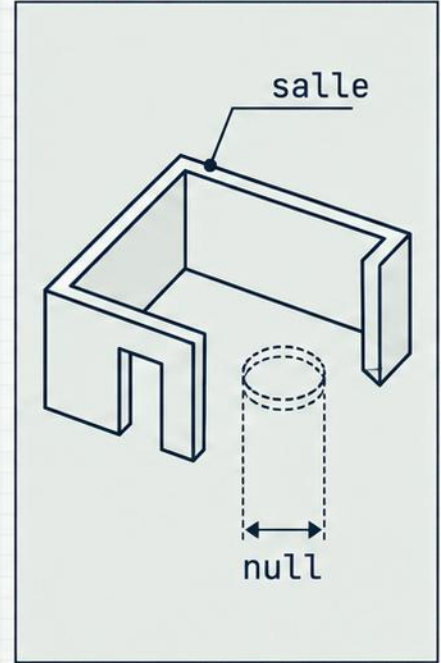
Le sosie porte le même nom et les mêmes points de vie, c'est pourtant un autre objet.
Pour savoir si deux variables désignent le même objet, prenez ===.

?-> : si loot vaut null, l'expression entière vaut null

```
1 $room = new Room('Trésor');  
2 var_dump($room->loot->name);
```

Warning: Attempt to read property "name" on null
NULL

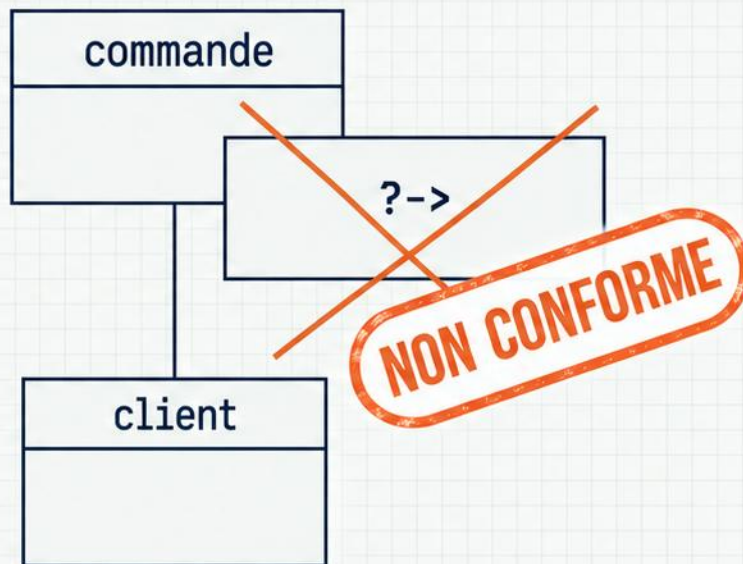
```
1 return $this->name.' : '.($this->loot ?->name ?? 'rien au sol');
```



- L'opérateur ?->, appelé nullsafe, évite le test. ?? fournit la valeur de remplacement. En trois caractères, vous évitez un if de trois lignes.

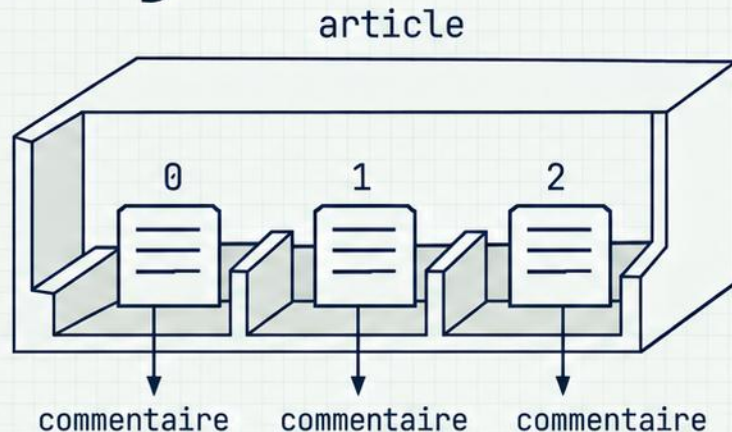
? -> convient pour un affichage, jamais pour cacher une erreur du métier

- Un prêt sans exemplaire, une commande sans client : l'absence est une erreur. Levez une exception, c'est le chapitre 6.
- Sinon l'erreur n'apparaît que beaucoup plus loin dans le programme, là où elle devient difficile à comprendre.



\$post->comments : la même structure que \$items dans Inventory

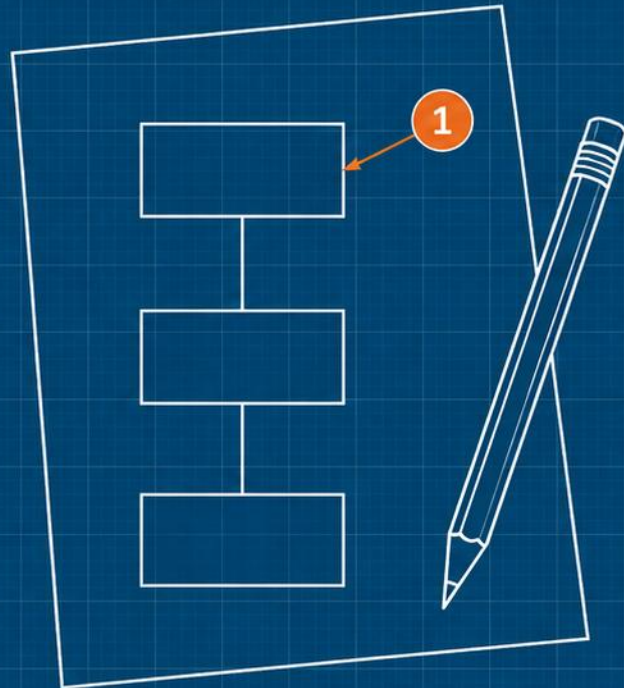
```
1 $post = Post::find(1);  
2  
3 foreach ($post->comments as $comment) {  
4     echo $comment->body;  
5 }
```



- `Post::find(1)` va chercher l'article numéro 1 dans la base de données et le rend sous forme d'objet. `$post->comments` donne la liste de ses objets `Comment`.
- Laravel remplit cette liste depuis la base de données, vous n'écrivez aucun `new`.
- `find()` renvoie `null` quand aucun article ne porte ce numéro :
`$post->comments ?? []`, ou `Post::findOrFail(1)` qui renvoie une page 404.

Exercices sur papier, avant d'ouvrir le kata

1. Un Dungeon reçoit ses salles en paramètre au lieu de les fabriquer. Quel losange dessinez-vous alors, et que se passe-t-il si deux donjons reçoivent la même salle ?
2. Reprenez le diagramme de la section 3. Écrivez la propriété PHP qui traduit chacune de ses trois relations à losange : la salle et son butin, le héros et son sac, le sac et ses objets. Donnez à chaque fois la visibilité, le type et la valeur de départ.
3. Le héros expose son sac par la propriété `inventory`. N'importe quelle partie du programme peut donc écrire `$arthur->inventory->add(...)` sans passer par le héros. Est-ce compatible avec la composition ? Que changeriez-vous pour que seul le héros remplisse son sac ?



Kata niveau 2 : Item, Inventory, Hero

```
composer test -- --group=niveau-2
```

- Item : constructeur avec validation du poids, un poids négatif lève une `InvalidArgumentException`.
- Inventory : `add()`, `has()`, `remove()`, `count()`, `totalWeight()`. Rien ne s'écrit de l'extérieur.
- Hero : `public readonly Inventory $inventory`, créé dans le constructeur. Deux héros n'ont jamais le même sac.

Le seul piège du niveau : si deux héros partagent un sac, le test passe au rouge.

