

Bloc P00 • chapitre 9

Atelier : recodez la Collection de Laravel

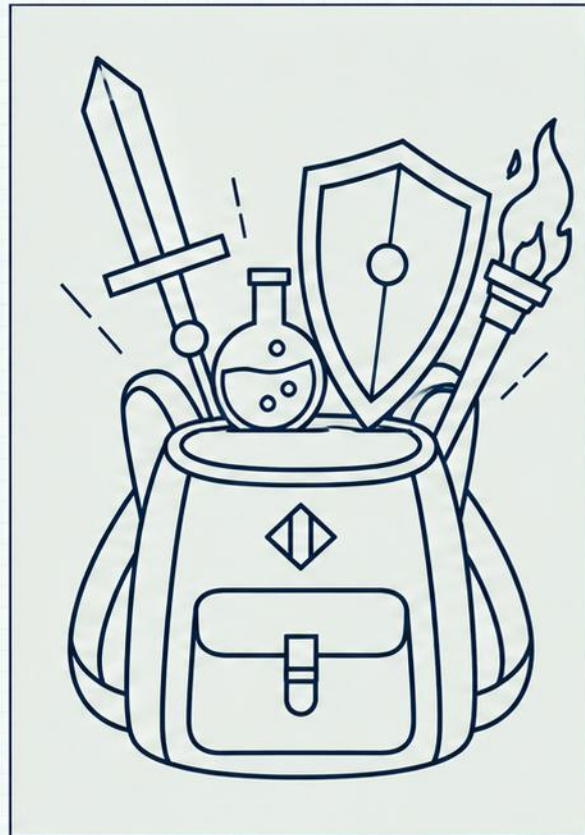
Une pile, une liste, une collection : trois classes qui font passer les tests, et la documentation de Laravel devient lisible.



Le sac d'Arthur en dix lignes : ce code sent mauvais

```
1  $items = [  
2      ['name' => 'Épée longue', 'weight' => 3.5],  
3      ['name' => 'Potion de soin', 'weight' => 0.5],  
4      ['name' => 'Bouclier', 'weight' => 6.0],  
5      ['name' => 'Torche', 'weight' => 1.0],  
6  ];  
7  
8  $total = 0;  
9  $noms = '';  
10  for ($i = 0; $i < count($items); $i++) {  
11      if ($items[$i]['weight'] < 5) {  
12          $total = $total + $items[$i]['weight'];  
13          $noms = $noms . $items[$i]['name'] . ', ';  
14      }  
15  }  
16  echo 'Léger : ' . $noms . '(' . $total . ' kg)', PHP_EOL;
```

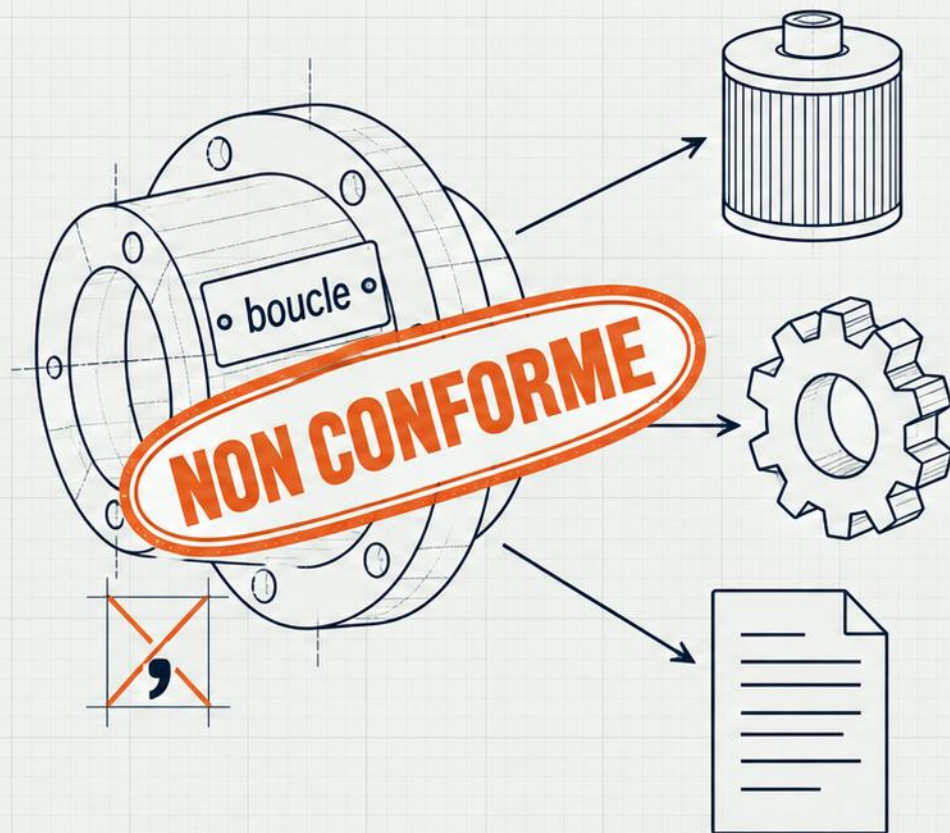
Léger : Épée longue, Potion de soin, Torche,(5 kg)



- Cinq minutes, en binôme. Le but est d'afficher les objets légers et leur poids total. Trouvez au moins trois choses à reprocher à cette boucle.

Trois métiers dans une boucle, une virgule en trop, un 5 venu de nulle part

- **La boucle fait trois métiers.**
Elle sélectionne, elle additionne et elle fabrique du texte. Trois raisons de la modifier, donc trois occasions de la casser.
- **La virgule en trop.** Regardez la sortie : Torche, (5 kg). C'est le défaut classique de la concaténation dans une boucle.
- **Le 5 sort de nulle part.**
Cinq quoi ? Un nombre sans nom, au milieu du code.



Dans Laravel, User::all() ne renvoie pas un tableau

```
1 $noms = $users->filter(fn ($user) => $user->isActive())  
2     ->sortBy(fn ($user) => $user->name)  
3     ->pluck('name');
```



- Il renvoie un objet, une `Illuminate\Support\Collection`.
Comme `$post->comments`, comme le résultat d'un `where()->get()`, comme presque tout ce qui contient plusieurs éléments.
- Pour lire ces trois lignes, il faut comprendre `filter()`, `sortBy()`, `pluck()` et la syntaxe `fn`.
- L'atelier construit cette `Collection` en trois étapes, puis compare son comportement à celui de Laravel.

Le dépôt : tout est rouge, une étape à la fois

```
git clone https://github.com/<votre-compte>/collection-26.git
cd collection-26
composer install
composer test
```

```
composer test -- --group=etape-1
composer test -- --group=etape-2
composer test -- --group=etape-3
composer test -- --group=bonus
```

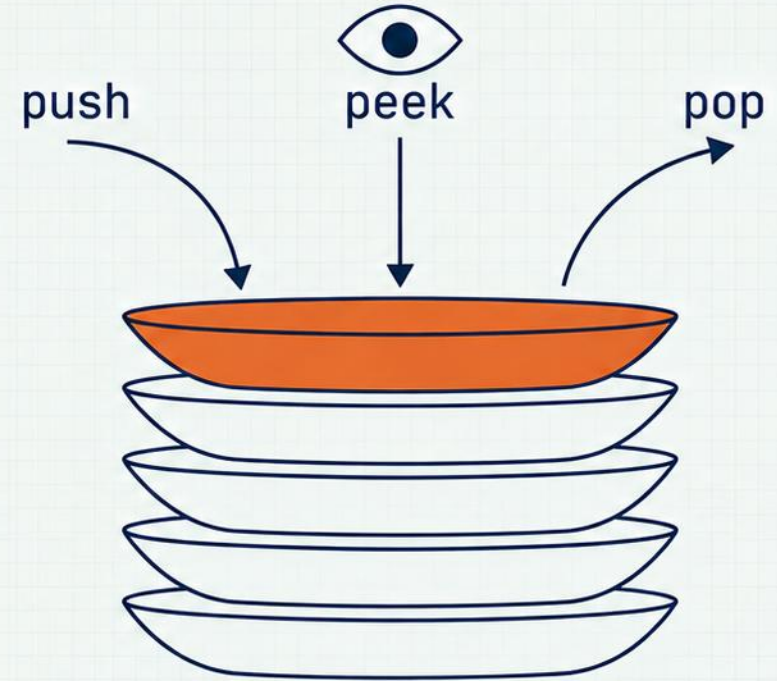


bonus

Chaque corps de méthode à compléter dans `src/` lève `LogicException('À implémenter')`. Les constructeurs, `guardIndex()` et la classe `Node` sont déjà écrits. Deux règles : ne modifiez pas les tests, ne touchez pas à `ListInterface`. Les tests sont le cahier des charges, l'interface est le contrat.

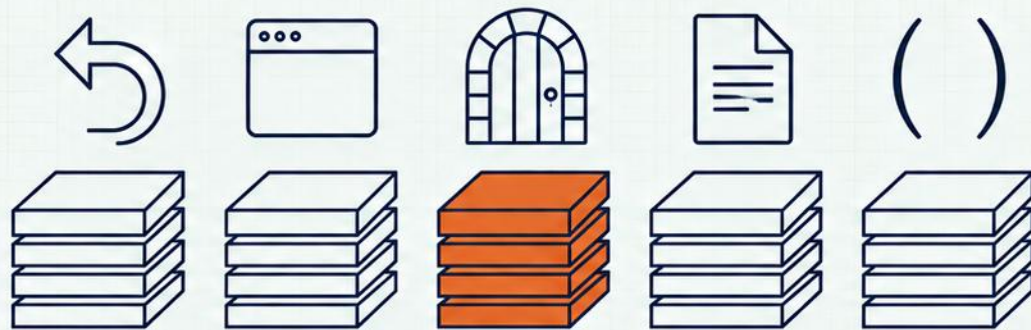
Étape 1 : une pile d'assiettes, le dernier arrivé est le premier servi

- Vous posez sur le dessus, vous reprenez sur le dessus. Pour atteindre l'assiette du milieu, il faut d'abord enlever celles d'au-dessus.
- C'est une pile **LIFO**, pour *last in, first out*.
- `push()` pose sur le dessus, `pop()` retire le dessus et le renvoie, `peek()` le consulte sans le retirer.
- `mixed` est le type PHP qui accepte n'importe quelle valeur : une pile ne regarde pas ce qu'elle range.



À quoi sert une pile : partout où l'on revient en arrière

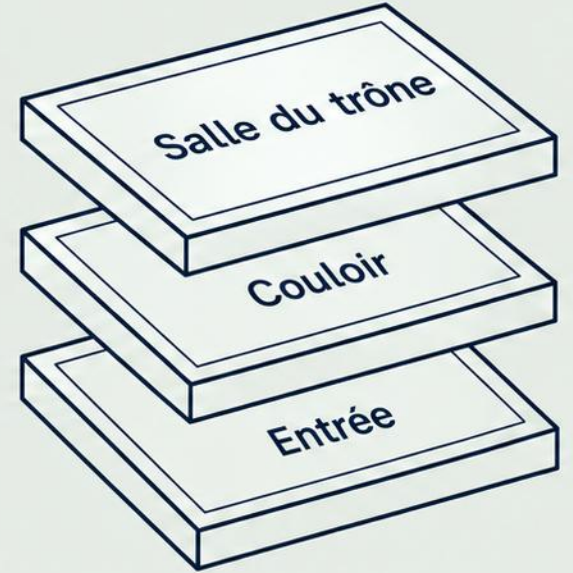
- **Annuler, Ctrl+Z** : chaque modification est empilée, annuler dépile la dernière.
- **Le bouton Précédent** du navigateur : les pages visitées, la dernière en haut.
- **Le Donjon** : l'historique des salles visitées. Faire demi-tour revient à reprendre la dernière.
- **PHP lui-même** : chaque appel de fonction est empilé. La « Stack trace » d'une erreur, c'est cette pile, lue du sommet vers la base.
- **Un éditeur de code** : une parenthèse ouverte s'empile, une fermée la dépile. Pile vide à la fin : tout est équilibré.



push(), pop(), peek(), isEmpty(), count() : cinq méthodes, un tableau

```
1 $visitees = new Stack();
2 $visitees->push('Entrée');
3 $visitees->push('Couloir');
4 $visitees->push('Salle du trône');
5
6 echo $visitees->peek(), PHP_EOL;
7 echo $visitees->pop(), PHP_EOL;
8 echo $visitees->pop(), PHP_EOL;
9 echo count($visitees), PHP_EOL;
```

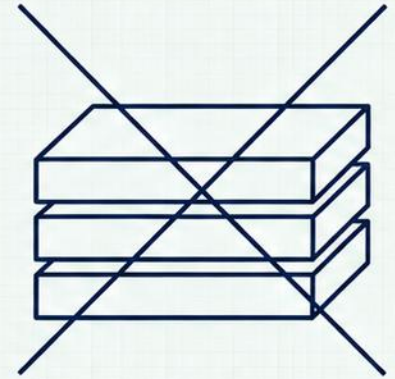
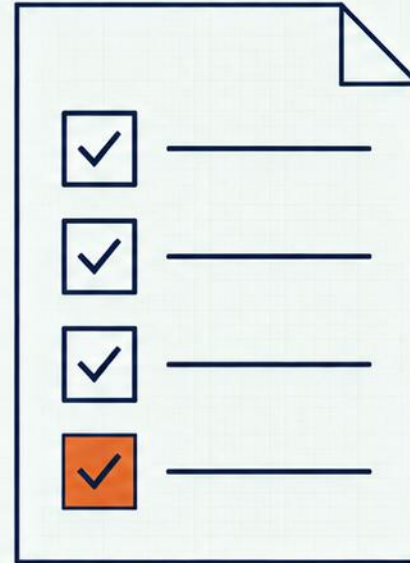
```
Salle du trône
Salle du trône
Couloir
1
```



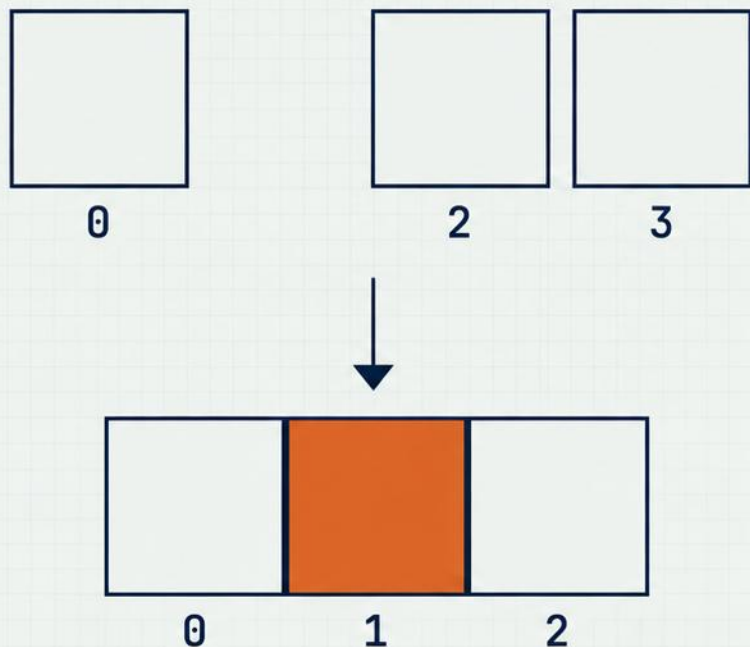
Salle du trône sort deux fois : peek() regarde, pop() retire. Votre classe déclare implements Countable : la fonction native count() appelle votre méthode count(), comme au chapitre 7.

Ce que les 9 tests de l'étape 1 attendent

- Une pile neuve est vide. Après trois `push()`, `count()` vaut 3.
- `pop()` rend les éléments dans l'ordre inverse de l'empilement, puis la pile redevient vide.
- `peek()` rend le sommet sans le retirer : deux appels de suite donnent la même valeur.
- Pile vide : `pop()` et `peek()` lèvent `UnderflowException`, une exception de PHP prête à l'emploi.
- **Indice** : tout se joue à la fin du tableau. `array_pop()` retire le dernier élément, `array_key_last()` donne son index sans y toucher.
- **Question** : pourquoi une pile n'a-t-elle pas de méthode `get(int $index)` ? Qu'est-ce que vous gagnez à interdire quelque chose ?



Étape 2 : ArrayList, une liste sans trou, et un contrat

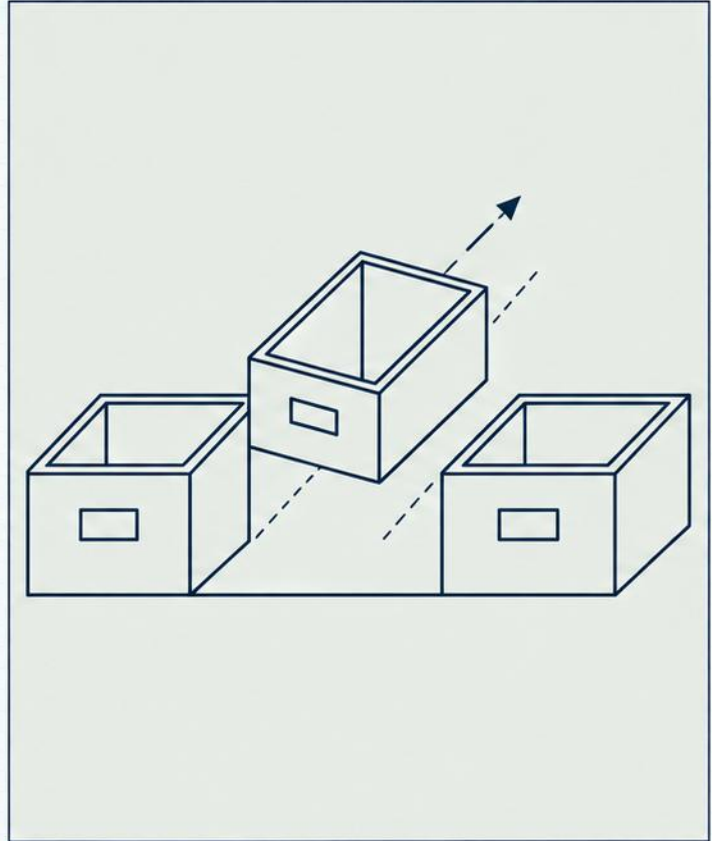


- Un tableau PHP garde des trous : après `unset($tab[1])`, les index restants sont 0, 2, 3.
- ArrayList est indexée de 0 à `size() - 1`, sans trou : après `remove(1)`, l'ancien index 2 devient l'index 1.
- Hors des bornes, `get()` lève `OutOfRangeException`, au lieu d'un simple avertissement et d'un `null`.
- `indexOf()` compare avec `===` : "2" n'est pas 2. Absent : -1.
- ListInterface dit ce qu'on peut faire, pas comment. Un paramètre typé ListInterface acceptera demain une autre mécanique.

ArrayList : onze méthodes, et remove() décale les suivants

```
1 $compagnie = new ArrayList();
2 $compagnie->push('Arthur');
3 $compagnie->push('Perceval');
4 $compagnie->push('Merlin');
5 $compagnie->remove(1);
6
7 echo $compagnie->get(1), PHP_EOL;
8 echo $compagnie->size(), PHP_EOL;
9 echo $compagnie->indexOf('Perceval'), PHP_EOL;
10 echo $compagnie, PHP_EOL;
```

```
Merlin
2
-1
[
    "Arthur",
    "Merlin"
]
```

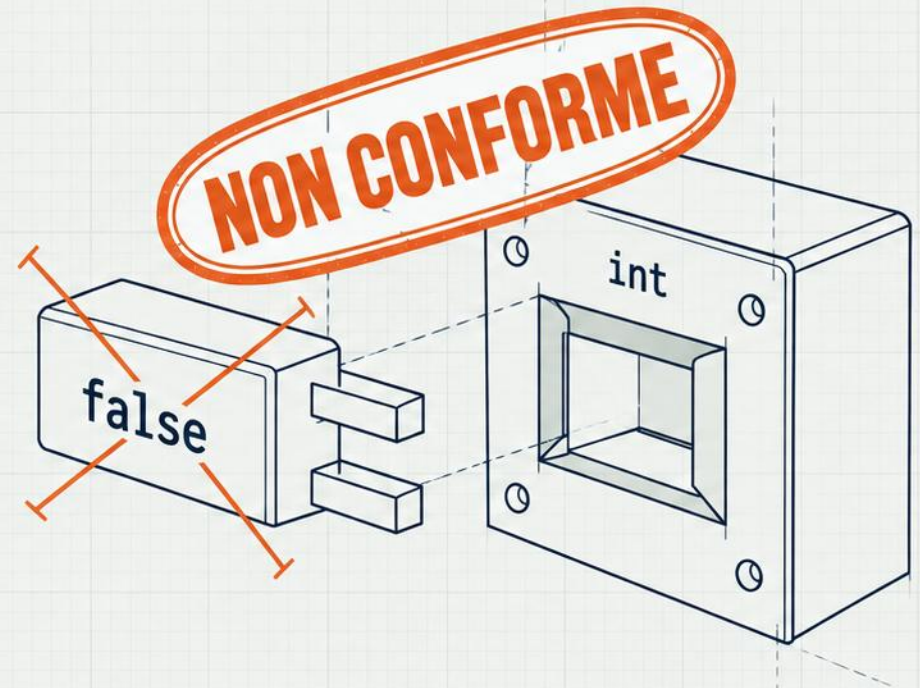


Perceval retiré : Merlin passe à l'index 1, et indexOf('Perceval') rend -1. __toString() rend exactement la sortie de json_encode() avec JSON_PRETTY_PRINT et JSON_UNESCAPED_UNICODE. **Indice** : array_splice() retire et renumérote. Appelez guardIndex() au début de get(), set() et remove().

Le piège de indexOf() : array_search() rend false, pas -1

Fatal error: Uncaught TypeError: Collections\ArrayList::indexOf(): Return value must be of type int, false returned

- `array_search($element, $this->elements, true)` cherche bien avec `===`, mais renvoie `false` quand l'élément est absent.
- La méthode est déclarée : `int` et le fichier `declare(strict_types=1)` : PHP refuse ce retour. C'est à vous de traduire le `false` en `-1`.
- Sans `declare(strict_types=1)`, PHP convertirait `false` en `0` sans rien dire, et votre liste répondrait que l'absent est en première position.

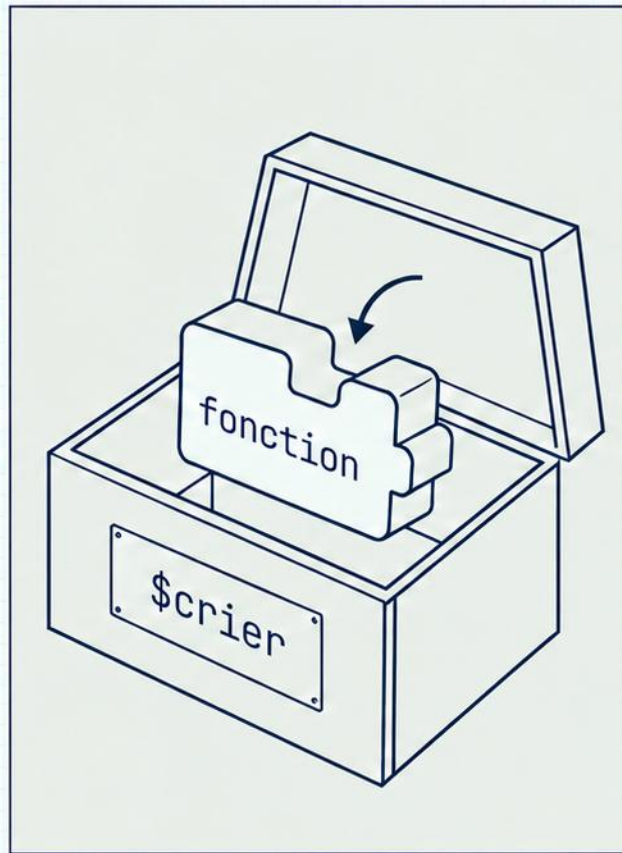


Une fonction est une valeur : on la range, on la passe en paramètre

```
1 $crier = fn(string $nom): string => mb_strtoupper($nom) . ' !';  
2  
3 echo $crier('Arthur'), PHP_EOL;
```

ARTHUR !

- Une fonction passée à une autre pour être appelée par elle s'appelle un **callback**.
- `fn (paramètres): type => expression` est la **fonction fléchée** : une seule expression, dont le résultat est renvoyé sans `return`. C'est la forme que vous verrez partout dans Laravel.
- `mb_strtoupper()` met en majuscules, lettres accentuées comprises.

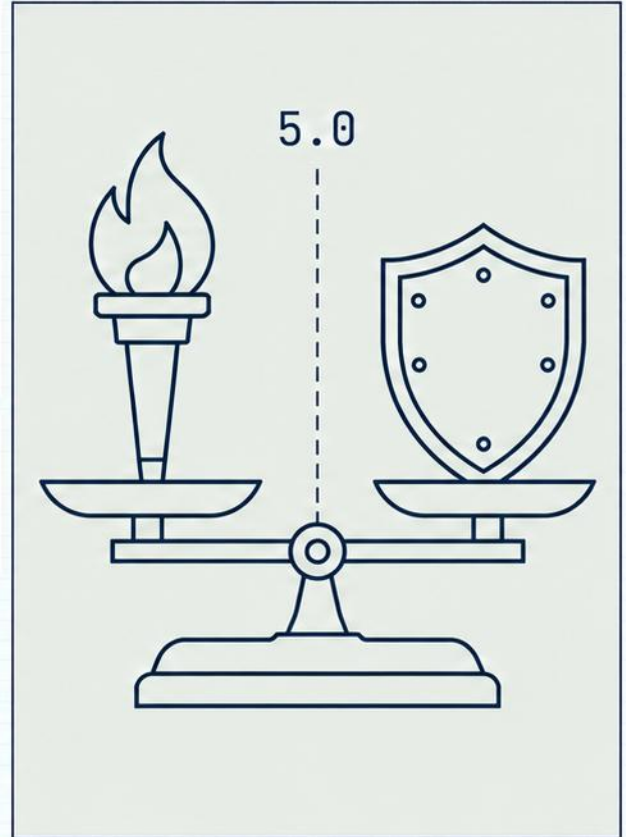


La fonction fléchée utilise les variables autour d'elle, sans les déclarer

```
1 $limite = 5.0;
2 $estLeger = fn (Item $item): bool => $item->weight < $limite;
3
4 echo $estLeger(new Item('Torche', 1.0)) ? 'oui' : 'non', PHP_EOL;
5 echo $estLeger(new Item('Bouclier', 6.0)) ? 'oui' : 'non', PHP_EOL;
```

oui
non

- `$limite` est défini à l'extérieur, la fonction fléchée le lit toute seule.
Item vient des tests du dépôt, `tests/Fixtures/Item.php` : un nom et un poids, tous deux public readonly. Dans une signature, le type d'un callback s'écrit callable.



Le piège du use oublié : fonction ne capture rien toute seule

```
1 $limite = 5.0;
2 $estLeger = function (Item $item): bool {
3     return $item->weight < $limite;
4 };
5
6 echo $estLeger(new Item('Torche', 1.0)) ? 'oui' : 'non', PHP_EOL;
```

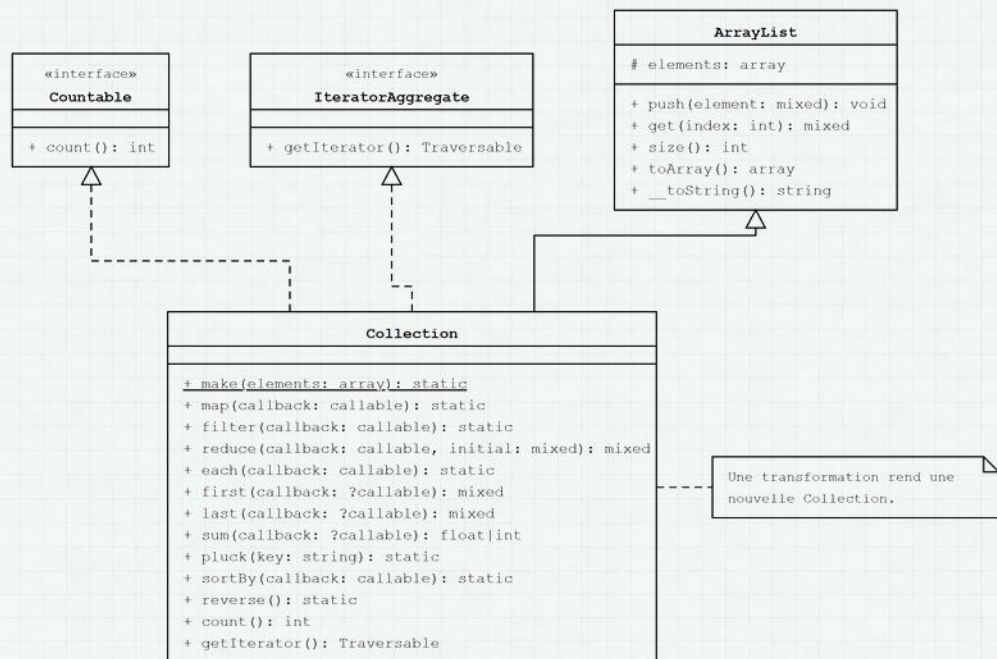
NON CONFORME

Warning: Undefined variable \$limite in leger.php on line 4
non

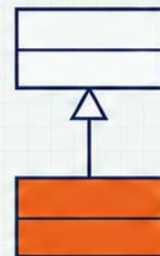
PHP se contente d'un avertissement, traite \$limite comme null, et répond non. Votre torche de 1 kg n'est plus légère. La version correcte s'écrit `function (Item $item) use ($limite): bool { ... }`. L'écriture longue reste utile quand le corps fait plusieurs lignes.



Étape 3 : Collection extends ArrayList, vous n'ajoutez que le neuf



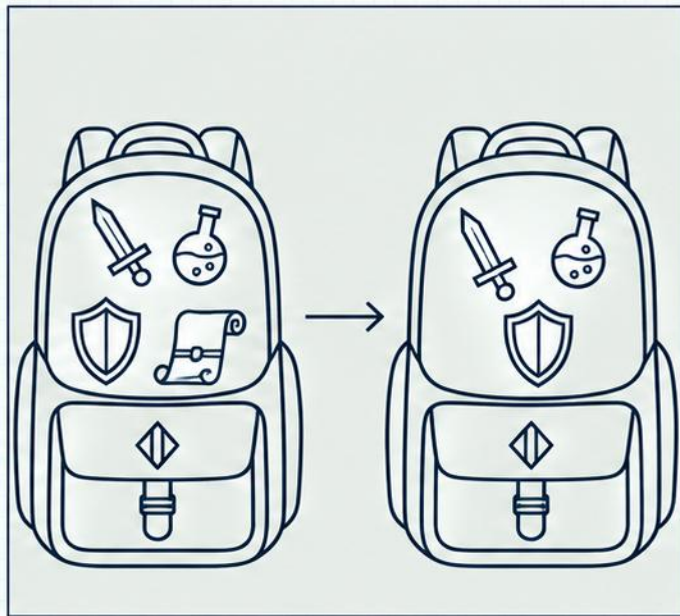
- Trait plein au triangle vide : push, get, size, toArray et __toString sont hérités. Rien à réécrire.
- Collection ajoute les transformations : map, filter, pluck, sortBy, reduce, sum, first, last...
- Countable donne count(\$collection), IteratorAggregate donne foreach, comme l'Inventory du chapitre 7.
- static dans une signature désigne la classe appelée. make() est la fabrique : comme Dice::d6() au chapitre 1, comme collect() dans Laravel. Écrivez-la en premier.



La règle : une transformation ne modifie jamais la collection de départ

```
1 $sac = Collection::make($butin);  
2 $legers = $sac->filter(fn (Item $item): bool => $item->weight < 5.0);  
3  
4 echo $sac->size(), PHP_EOL;  
5 echo $legers->size(), PHP_EOL;
```

4
3



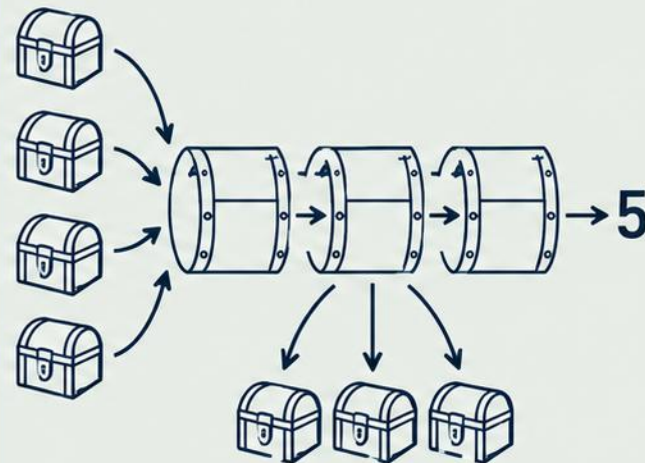
Le sac d'origine n'a pas bougé. C'est ce qui rend le chaînage sûr : chaque maillon reçoit une collection et en produit une autre, sans modifier la précédente. `each()` est la seule exception : elle exécute le callback pour son effet, puis rend la collection elle-même.

filter(), pluck(), sum() : chaque ligne fait une seule chose

```
1 $noms = Collection::make($butin)
2   ->pluck('name')
3   ->map(fn (string $nom): string => mb_strtoupper($nom));
4
5 echo $noms->get(0), PHP_EOL;
6
7 $poids = Collection::make($butin)
8   ->filter(fn (Item $item): bool => $item->weight < 5.0)
9   ->pluck('weight')
10  ->sum();
11
12 echo $poids, PHP_EOL;
```

ÉPÉE LONGUE

5

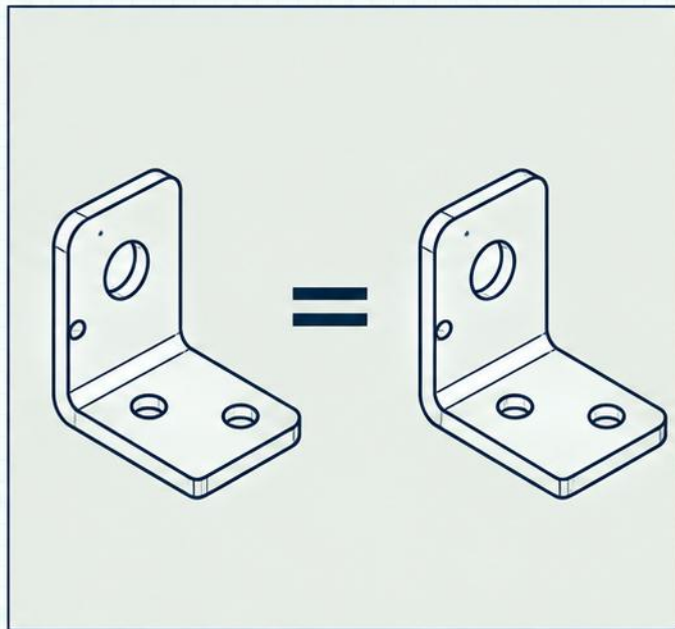


Quatre lignes pour le poids des objets légers : sélectionner, extraire, additionner. `filter()` sélectionne, il ne trie pas : c'est `sortBy()` qui trie. `pluck('name')` lit la propriété `name` de chaque objet.

Côte à côte : votre Collection, puis celle de Laravel

```
1 // Votre Collection
2 echo Collection::make($butin)
3     ->filter(fn (Item $item): bool => $item->weight < 5.0)
4     ->sum(fn (Item $item): float => $item->weight), PHP_EOL;
5
6 // Illuminate\Support\Collection
7 echo collect($butin)
8     ->filter(fn (Item $item): bool => $item->weight < 5.0)
9     ->sum(fn (Item $item): float => $item->weight), PHP_EOL;
```

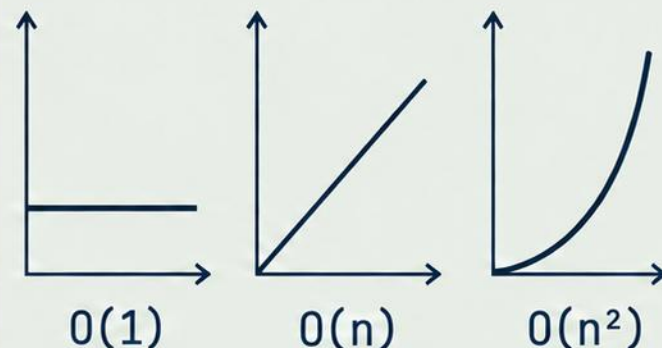
5
5



- Mêmes noms de méthodes, mêmes callbacks, même règle de la nouvelle collection. `collect()` est un raccourci pour `new Collection(...)`, l'équivalent de votre `make()`.
- Une vraie différence : le `filter()` de Laravel conserve les clés d'origine, 0, 1 et 3. D'où le `->values()` que vous croiserez souvent : il remet les index à plat.
- La vraie Collection a plus de cent cinquante méthodes de plus. Vous avez de quoi les comprendre en lisant leur description.

Le coût d'un algorithme : Big O compte des étapes, pas des secondes

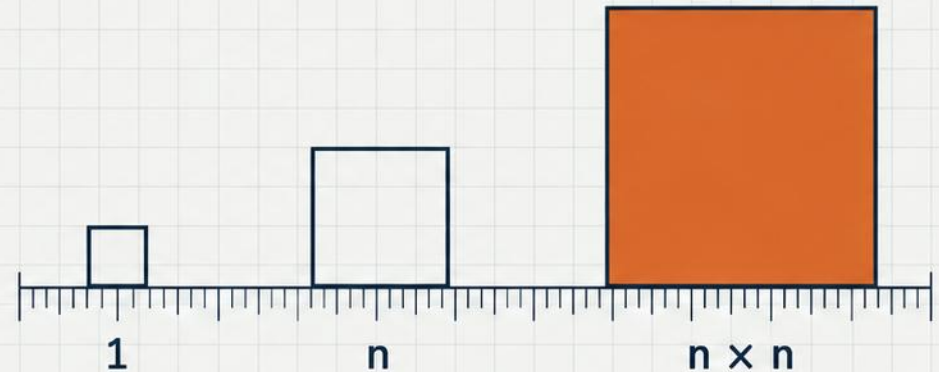
```
1 $noms = ['Arthur', 'Perceval', 'Merlin', 'Guenièvre'];
2
3 $premier = $noms[0];           // 0(1) : une étape, quelle que soit la taille
4
5 foreach ($noms as $nom) {      // 0(n) : 4 étapes pour 4 éléments
6 }
7
8 foreach ($noms as $a) {        // 0(n²) : 16 étapes pour 4 éléments
9     foreach ($noms as $b) {
10 }
11 }
```



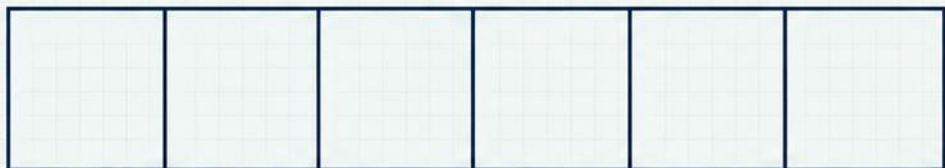
La notation ne garde que ce qui domine quand n grandit. Pour 4 noms : 1 étape, 4 étapes, 16 étapes. Pour 1 000 noms : 1, 1 000, et 1 000 000.

Compter les étapes soi-même : quatre règles

- Une instruction simple, un accès par index : **1 étape**, quelle que soit la taille. $O(1)$: `get()`, `push()`, `size()`, `Stack::peek()`.
- Une boucle qui parcourt les n éléments : **n étapes**. $O(n)$: `indexOf()`, `includes()`, `map()`, `filter()`, `sum()`.
- Une boucle dans une boucle : **$n \times n$ étapes**. $O(n^2)$: une boucle sur une liste qui appelle `indexOf()` à chaque tour.
- On oublie les constantes et on garde le terme qui domine : $2n$ étapes, c'est encore $O(n)$. $n^2 + n$, c'est $O(n^2)$.
- Le piège : `remove()` sur `ArrayList` est $O(n)$, parce qu'`array_splice()` renumérote tout ce qui suit.



Bonus : même contrat, autre mécanique, autre coût



- Chaque maillon, un Node, garde un élément et le maillon suivant. Les éléments ne sont plus rangés côte à côte.
- Lire le 50^e élément demande de traverser les 49 précédents : `get()` passe de $O(1)$ à $O(n)$.
- Retirer un maillon déjà atteint ne décale rien : le précédent reçoit le suivant dans `next`. Mais `remove(index)` reste $O(n)$, il faut d'abord l'atteindre.
- Un paramètre typé `ListInterface` accepte une `ArrayList` comme une `LinkedList`, alors que tout diffère à l'intérieur.

À vous : trois étapes vertes, puis le sac d'Arthur en cinq lignes

Sur papier

1. Sans lancer le code, écrivez la sortie de `Collection::make($butin)->reverse()->first()->name` , puis de `Collection::make($butin)->sortBy(fn (Item $i): float => $i->weight)->last()->name` .
2. Classez en `O(1)` , `O(n)` ou `O(n2)` : `Stack::peek()` , `ArrayList::includes()` , `LinkedList::get()` , et une boucle sur une liste qui appelle `includes()` sur une autre liste à chaque tour.



Atelier

1. **Obligatoire.** Les groupes `etape-1` à `etape-3` verts, poussés, vérification verte dans l'onglet Actions.
2. **Obligatoire.** `examples/ouverture.php` : le code de la slide 2 réécrit avec votre `Collection` , cinq lignes, une constante pour le `5` , une virgule au bon endroit.
3. **Au choix.** `avg()` , la lecture de <https://laravel.com/docs/collections>, ou `composer require --dev illuminate/collections` pour lancer les deux chaînes de la slide 19.
4. **Bonus.** `LinkedList` , groupe `bonus` .

