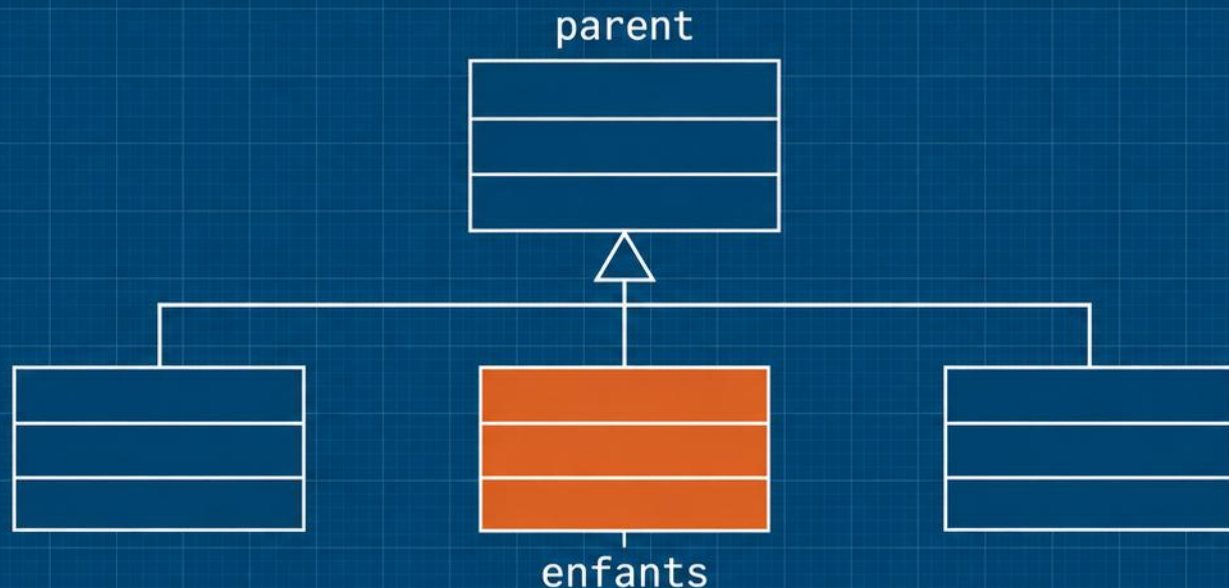


## Bloc P00 · chapitre 6

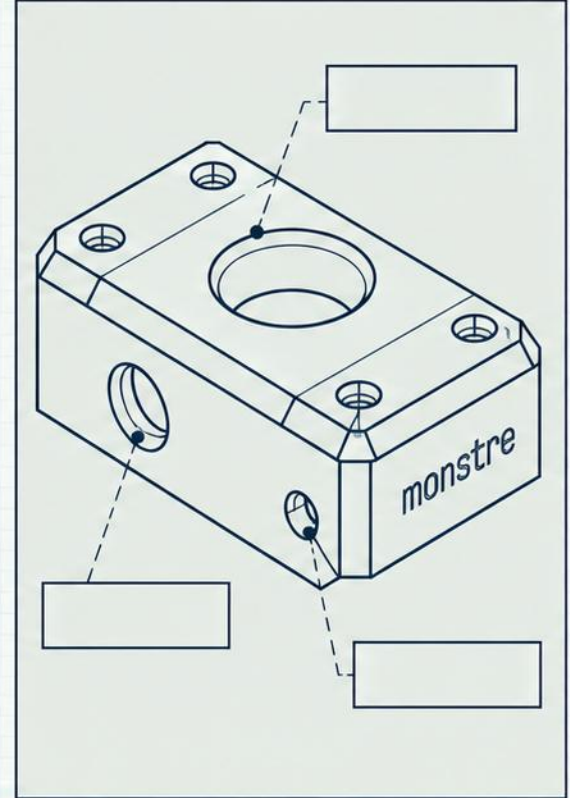
# Héritage et polymorphisme

Le code commun à un seul endroit, et trente monstres différents sans un seul if.



# Que faudra-t-il modifier pour ajouter un squelette ?

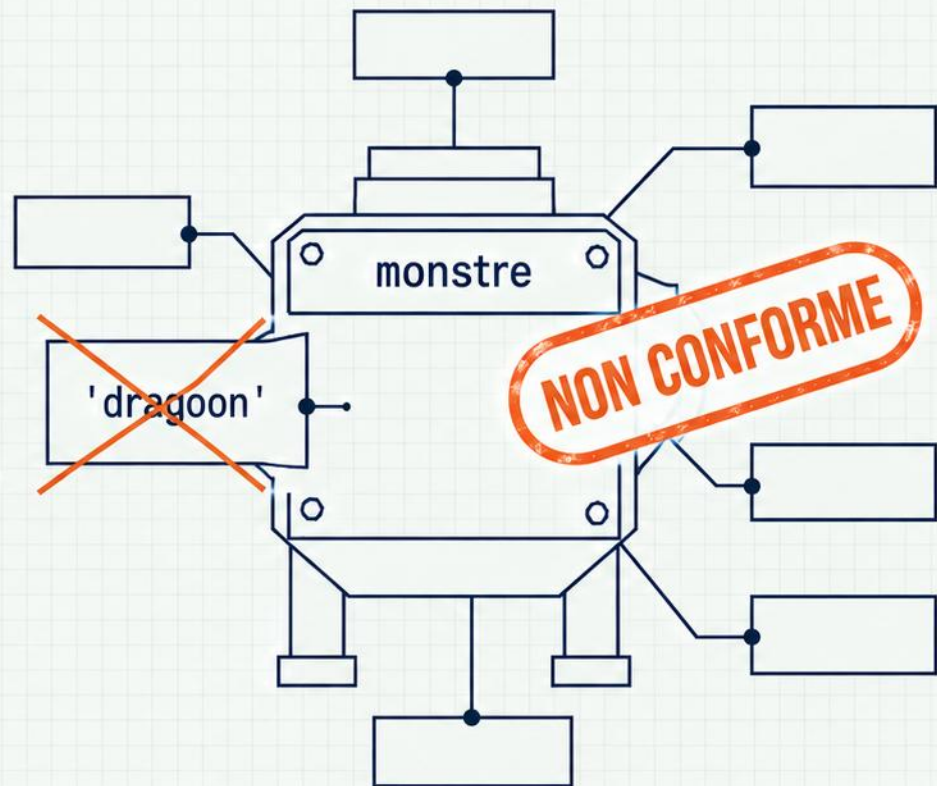
```
1  class Monster
2  {
3      public function __construct(private string $type, private int $hp) {}
4
5      public function attack(): int
6      {
7          if ($this->type === 'goblin') {
8              return 2;
9          } elseif ($this->type === 'dragon') {
10             return 8;
11          }
12
13         return 0;
14     }
15 }
```



- Cinq minutes, en binôme. Ce code sent mauvais : un détail d'écriture y trahit un défaut de conception.

# Cette méthode, et toutes celles qui suivront

- `describe()`, `sound()` et toutes celles que le jeu inventera : chacune répétera la même cascade de `elseif`.
- Le type est une chaîne de caractères, donc `new Monster('dragoon', 30)` s'écrit sans erreur et `attack()` renvoie 0 en silence.
- Une seule classe porte les règles de tous les monstres du jeu.



La réponse du chapitre : **une classe par monstre, une méthode `attack()` chacune.**

# Deux classes complètes : le nom, le poids et sa validation écrits deux fois

```
1 final class Weapon
2 {
3     public function __construct(
4         public readonly string $name,
5         public readonly float $weight,
6         public readonly int $damage,
7     ) {
8         if ($weight < 0) {
9             throw new \InvalidArgumentException("Un poids n'est pas négatif, $weight reçu.");
10        }
11    }
12 }
13
```

```
14 final class Potion
15 {
16     public function __construct(
17         public readonly string $name,
18         public readonly float $weight,
19         public readonly int $healing,
20     ) {
21         if ($weight < 0) {
22             throw new \InvalidArgumentException("Un poids n'est pas négatif, $weight reçu.");
23        }
24    }
25 }
26
```

Deux problèmes. La duplication : le jour où un objet gagne une durabilité, vous modifiez deux fichiers, puis trois. Et sans classe commune, `Inventory::add()` doit énumérer les classes acceptées.



name, weight



name, weight

# Item porte le commun, Weapon ne décrit que sa différence

```
1  abstract class Item
2  {
3      public function __construct(
4          public readonly string $name,
5          public readonly float $weight,
6      ) {
7          if ($weight < 0) {
8              throw new \InvalidArgumentException("Un poids n'est pas négatif, $weight reçu.");
9          }
10     }
11
12     /** Chaque objet se décrit à sa façon : le parent ne peut pas répondre. */
13     abstract public function describe(): string;
14 }
```

```
15
16  final class Weapon extends Item
17  {
18      public function __construct(string $name, float $weight, public readonly int $damage)
19      {
20          parent::__construct($name, $weight);
21      }
22
23      public function describe(): string
24      {
25          return sprintf('%s : arme (%s kg, %d dégâts)', $this->name, $this->weight, $this->damage);
26      }
27  }
28
```

**extends** déclare que Weapon est un cas particulier d'Item. Elle en hérite toutes les propriétés et méthodes non privées : name, weight et la validation du poids, écrites une seule fois.

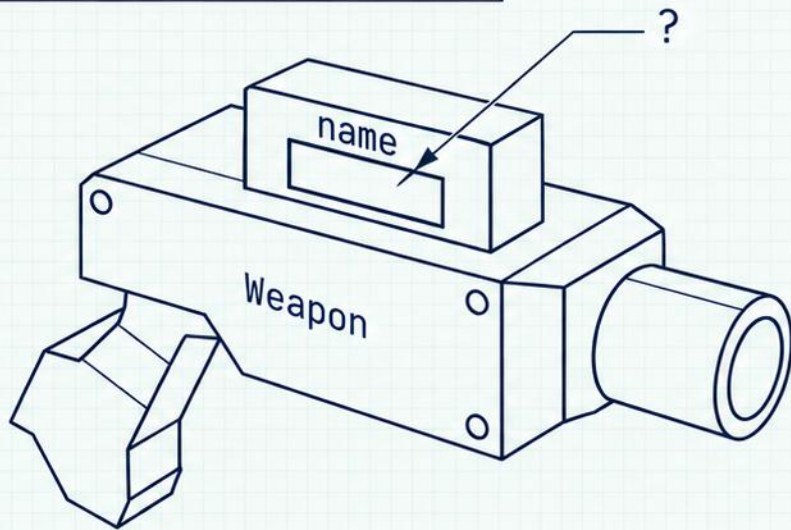


# Une classe fille qui définit son constructeur appelle celui du parent elle-même

```
parent::__construct($name, $weight);
```

Fatal error: Uncaught Error: Typed property Item::\$name must not be accessed before initialization

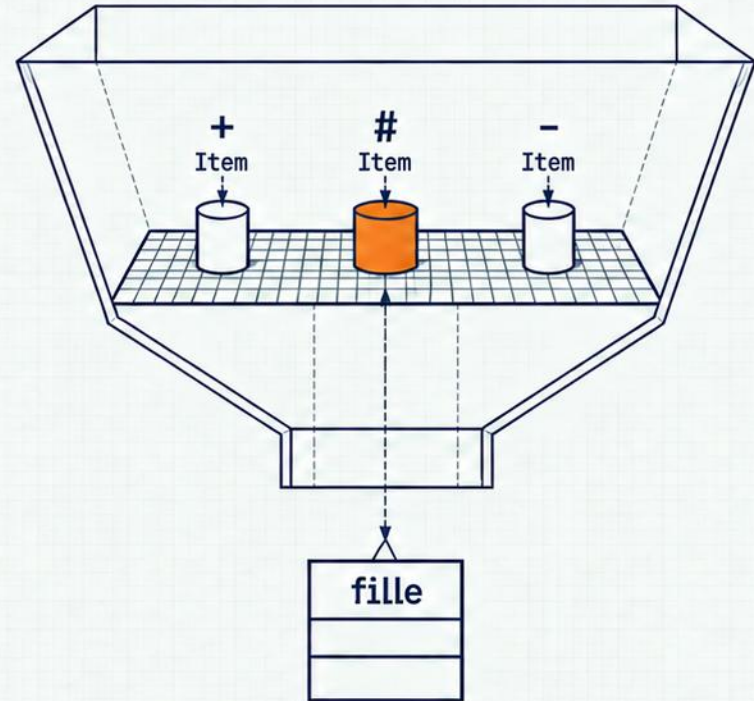
- `parent::__construct($name, $weight)` appelle le constructeur du parent.
- Sans lui, les propriétés du parent ne reçoivent aucune valeur, et la première lecture arrête le programme avec l'erreur ci-dessus.
- C'est l'erreur la plus fréquente du chapitre. Une classe fille sans constructeur utilise directement celui du parent.



# protected : la troisième visibilité, ouverte aux classes filles

- `public` : accessible n'importe où.  
C'est le mode d'emploi de la classe.
- `protected` : accessible dans la classe **et ses filles**. À utiliser quand les filles doivent écrire dans la donnée.
- `private` : la classe seule.  
Même les filles n'y touchent pas.

Ici, `Weapon::describe()` ne fait que lire `name` et `weight` : les deux restent `public readonly`. Aucune classe fille n'a besoin d'y écrire, donc `protected` ne sert pas.

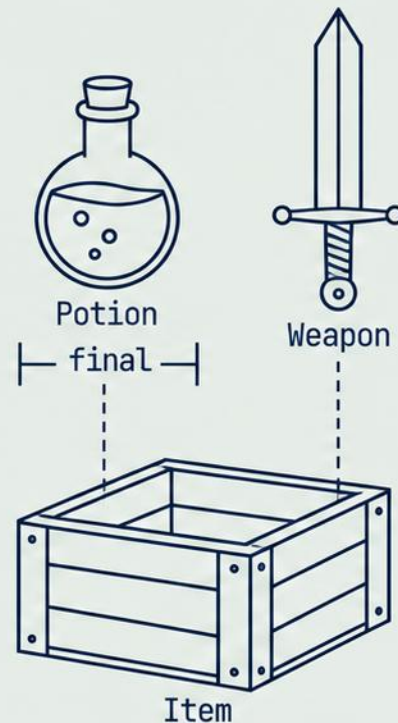


**Choisissez toujours la visibilité la plus fermée qui suffit.**

# Potion est bâtie sur le même modèle, avec healing à la place de damage

```
1  final class Potion extends Item
2  {
3      public function __construct(string $name, float $weight, public readonly int $healing)
4      {
5          parent::__construct($name, $weight);
6      }
7
8      public function describe(): string
9      {
10         return sprintf('%s : potion (%s kg, +%d PV)', $this->name, $this->weight, $this->healing);
11     }
12 }
13
14 echo (new Weapon('Épée courte', 2.0, 5))->describe(), PHP_EOL;
15 echo (new Potion('Potion de soin', 0.5, 5))->describe(), PHP_EOL;
```

Épée courte : arme (2 kg, 5 dégâts)  
Potion de soin : potion (0.5 kg, +5 PV)



`final` devant une classe interdit d'écrire une classe qui en hérite. Mettez-le sur les classes du bas de la hiérarchie, comme `Weapon` et `Potion`.

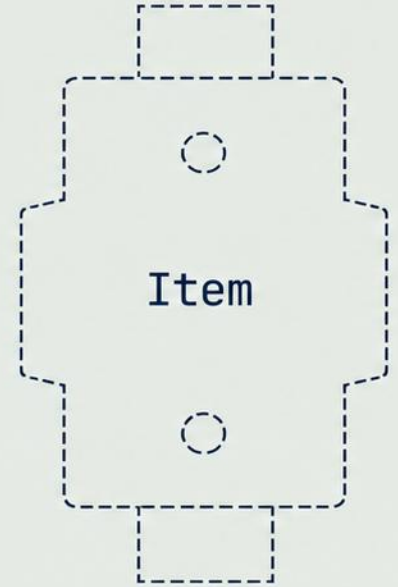
# abstract : une classe qui ne se fabrique pas, une méthode sans corps

```
1 $item = new Item('Caillou', 1.0);
```

Fatal error: Uncaught Error: Cannot instantiate abstract class Item

```
1 final class Rock extends Item
2 {
3 }
```

Fatal error: Class Rock contains 1 abstract method and must therefore be declared abstract or implement the remaining method (Item::describe)



abstract class Item : cette classe **ne s'instancie pas**. Elle n'existe que pour être héritée.

abstract public function describe(): string; : une **signature sans corps**. Toute classe fille concrète doit la fournir, sinon PHP refuse de charger le fichier.

# Monster porte les points de vie et laisse attack() abstraite

```
1  abstract class Monster
2  {
3      public private(set) int $maxHp;
4
5      public private(set) int $hp = 0 {
6          set => max(0, min($this->maxHp, $value));
7      }
8
9      public function __construct(
10         public readonly string $name,
11         int $maxHp,
12     ) {
13         $this->maxHp = $maxHp;
14         $this->hp = $maxHp;
15     }
16
17     public function takeDamage(int $amount): void
18     {
19         $this->hp -= $amount;
20     }
21
22     public function isAlive(): bool
23     {
24         return $this->hp > 0;
25     }
26
27     /** Chaque monstre frappe à sa façon. Le parent ne peut pas décider. */
28     abstract public function attack(): int;
29
30     public function __toString(): string
31     {
32         return sprintf('%s (%d/%d PV)', $this->name, $this->hp, $this->maxHp);
33     }
34 }
```

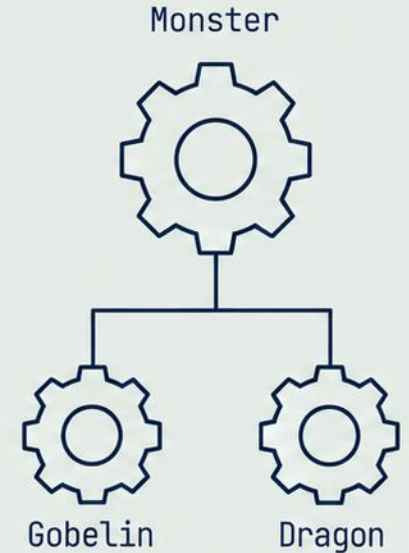


Des propriétés, un constructeur, des méthodes complètes, et une seule méthode abstraite. C'est ce qui distingue une classe abstraite d'une interface, au chapitre 7.

# Goblin et Dragon ne contiennent que leur différence

```
1 final class Goblin extends Monster
2 {
3     public function __construct()
4     {
5         parent::__construct('Goblin', 5);
6     }
7
8     public function attack(): int
9     {
10        return 2;
11    }
12 }
```

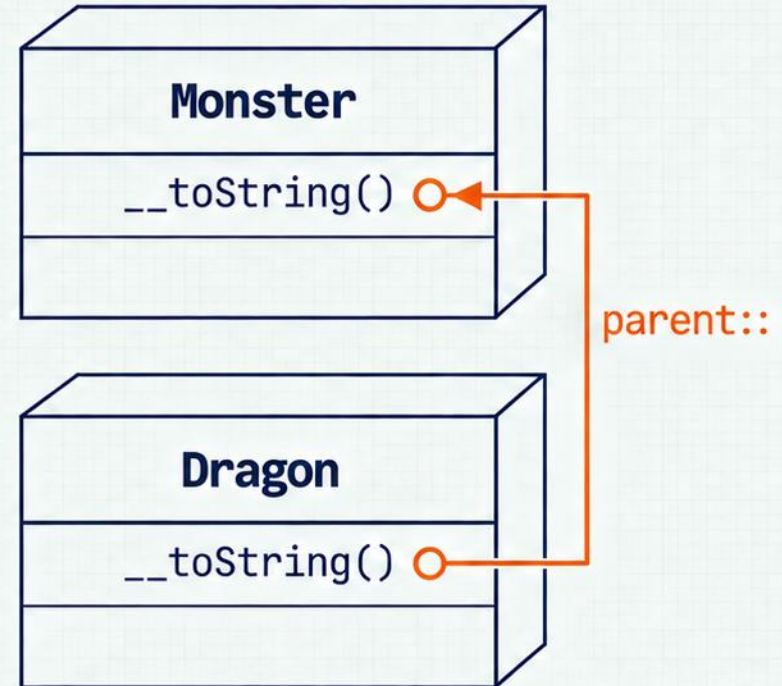
```
1 final class Dragon extends Monster
2 {
3     public function __construct()
4     {
5         parent::__construct('Dragon', 30);
6     }
7
8     public function attack(): int
9     {
10        return 8;
11    }
12 }
```



Les attaques valent 2 et 8, sans hasard : un monstre qui lance un dé est amusant, mais un test ne peut plus prévoir le résultat. Pas de protected : Goblin et Dragon n'écrivent jamais dans les points de vie, le parent s'en charge.

# Redéfinir : la même signature, et parent:: pour compléter au lieu de remplacer

- Une classe fille peut **redéfinir** une méthode du parent : elle écrit une méthode du même nom, et ce code remplace celui du parent pour les objets de la classe fille.
- La **signature** d'une méthode est sa ligne de déclaration : le nom, les paramètres avec leurs types, le type de retour. `public function __toString(): string` est la signature de `__toString()`.
- La méthode redéfinie garde la signature du parent : mêmes paramètres, même type de retour. PHP refuse le fichier sinon.
- `parent::` désigne la classe parente. `parent::__toString()` exécute la version de `Monster`, sur le même objet : `$this` y désigne toujours le dragon.



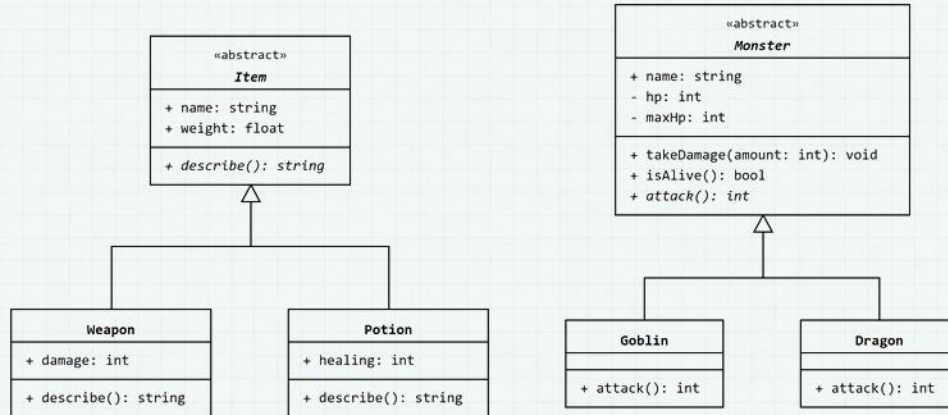
# Dragon complète le `__toString()` de `Monster` sans le recopier

```
1 final class Dragon extends Monster
2 {
3     // ... constructeur et attack() inchangés
4
5     public function __toString(): string
6     {
7         return 'Un dragon ! ' . parent::__toString();
8     }
9 }
10
11 echo new Goblin(), PHP_EOL;
12 echo new Dragon(), PHP_EOL;
```

Gobelin (5/5 PV)  
Un dragon ! Dragon (30/30 PV)

Sans `parent::`, `Dragon::__toString()` aurait recopié le `sprintf()` de `Monster`.  
Le jour où le format change, il change à un seul endroit.

# La généralisation : un trait plein terminé par un triangle vide, collé au parent

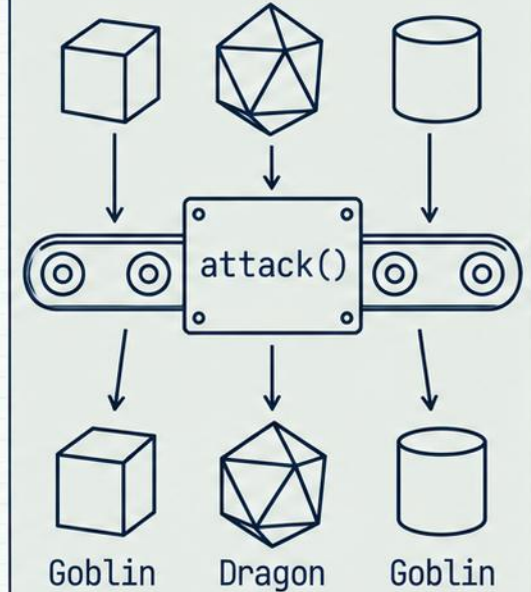


- Le trait se lit du bas vers le haut :  
« Weapon est un Item ».
- Le stéréotype «abstract» s'écrit au-dessus du nom, comme au chapitre 4. Le nom et la méthode abstraite sont *en italique*.
- Les enfants d'un même parent se rejoignent sur un seul triangle.

# Une liste de monstres de types différents, un seul appel de méthode

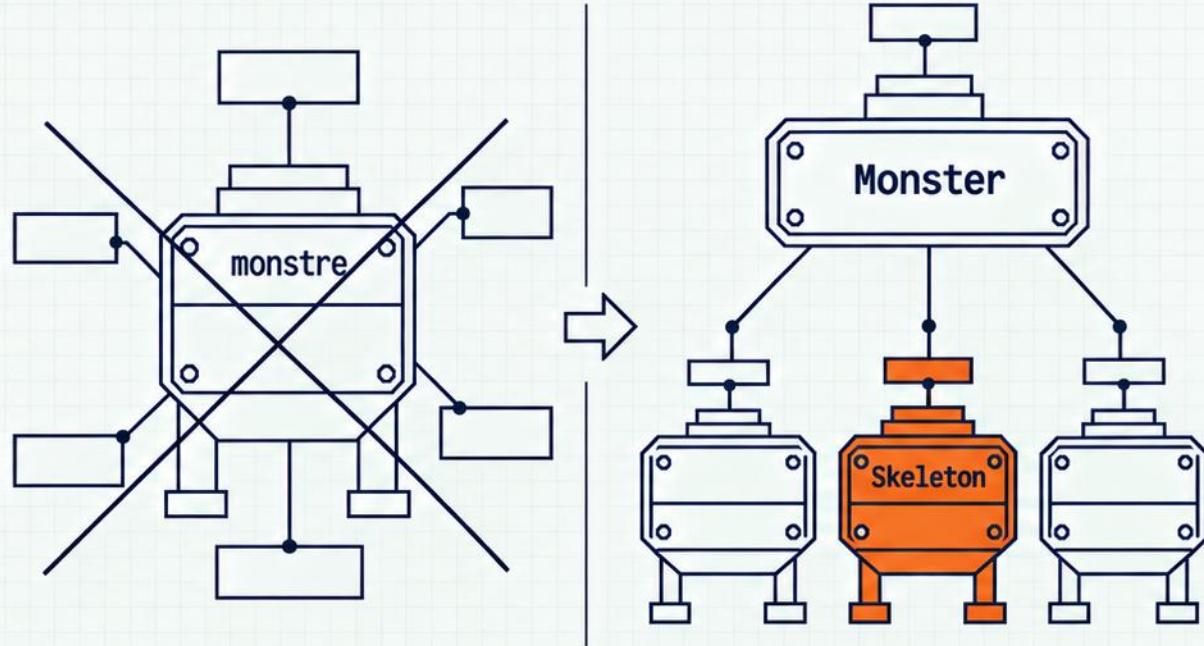
```
1  /** @var Monster[] $monsters */  
2  $monsters = [new Goblin(), new Dragon(), new Goblin()];  
3  
4  $total = 0;  
5  foreach ($monsters as $monster) {  
6      $damage = $monster->attack();  
7      $total += $damage;  
8      printf('%s frappe pour %d.%s', $monster->name, $damage, PHP_EOL);  
9  }  
10 printf('Total encaissé : %d PV.%s', $total, PHP_EOL);
```

Gobelin frappe pour 2.  
Dragon frappe pour 8.  
Gobelin frappe pour 2.  
Total encaissé : 12 PV.



# La cascade de `elseif` n'a pas été déplacée : elle a disparu

- Chaque monstre porte sa propre réponse à `attack()`, et la boucle ne connaît que le type `Monster`.
- Le vrai gain se voit à l'ajout d'un monstre. Vous créez `Skeleton` extends `Monster`, vous écrivez son `attack()`, et aucune boucle existante ne change.
- Comparez avec le code du début de chapitre : même résultat, aucun `if`.

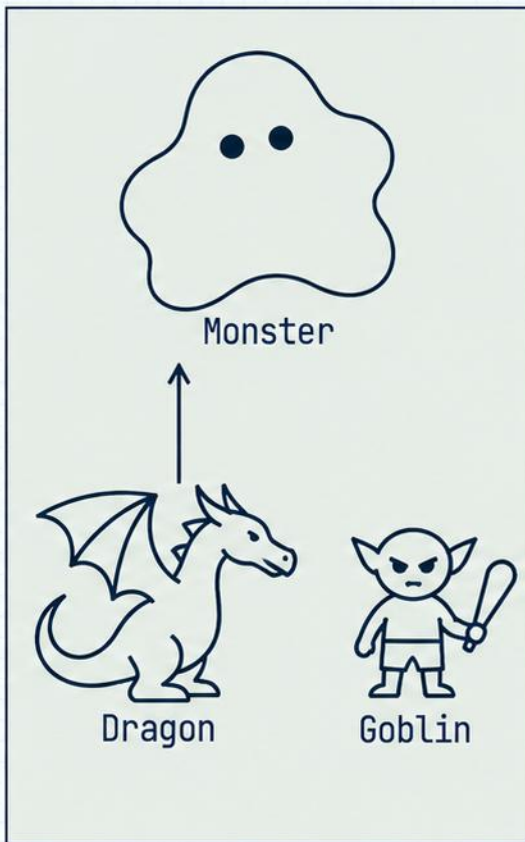


# C'est l'objet qui décide, pas la variable

```
1 $monster = new Dragon();  
2 echo get_class($monster), ' attaque pour ', $monster->attack(), PHP_EOL;  
3 var_dump($monster instanceof Monster, $monster instanceof Goblin);
```

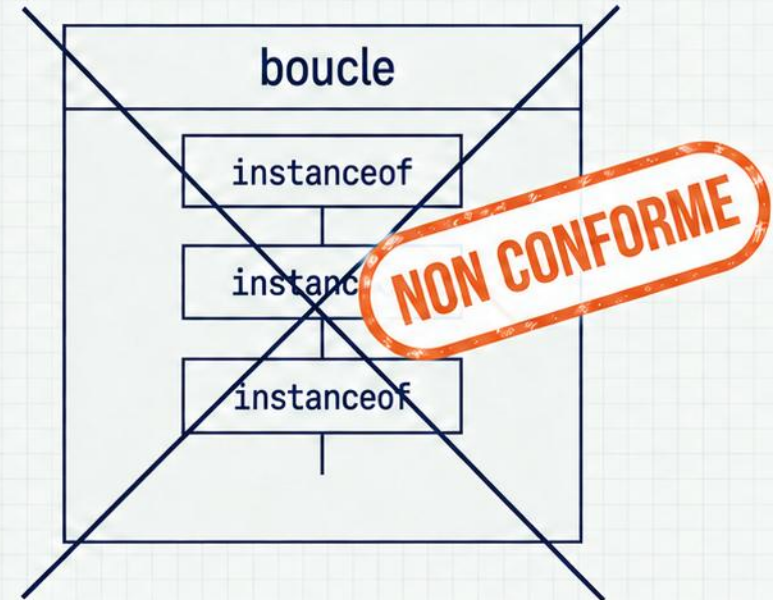
```
Dragon attaque pour 8  
bool(true)  
bool(false)
```

- `get_class()` rend le nom de la classe réelle de l'objet.
- À l'exécution, PHP part de la classe réelle de l'objet, ici Dragon, y cherche `attack()`, la trouve et l'appelle. S'il ne l'avait pas trouvée, il serait remonté à Monster. La recherche part toujours du bas.



# Une cascade de `instanceof`, c'est le `elseif` du début qui revient sous un autre nom

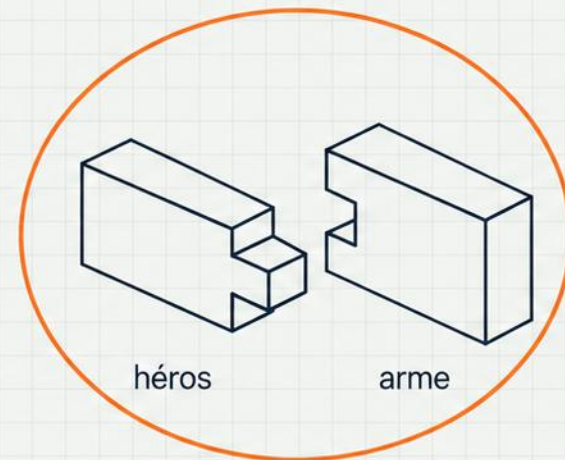
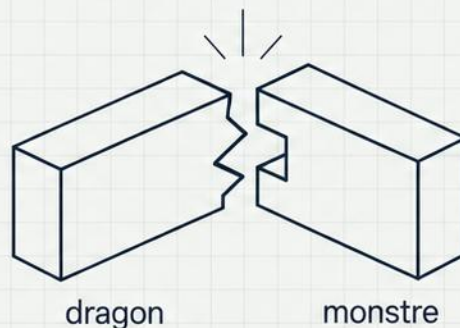
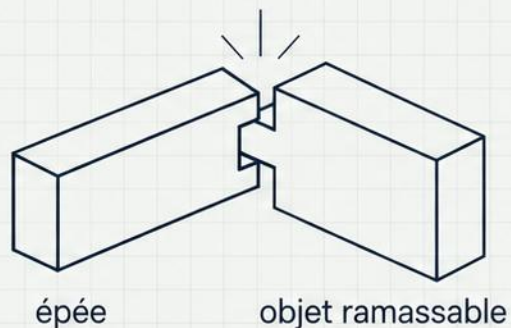
- `instanceof` répond à la question « cet objet est-il de ce type, ou d'un type qui en descend ? ». Il est utile pour une vérification ponctuelle.
- Une cascade de `instanceof` qui choisit l'attaque ramène le problème du début de chapitre.
- La réponse reste la même : une méthode commune aux monstres.



# Avant d'écrire extends, dites la phrase à voix haute : « un X est un Y »

- « Une épée **est un** objet ramassable. » Juste. Weapon extends Item.
- « Un dragon **est un** monstre. » Juste. Dragon extends Monster.
- « Un héros **est une** arme. » Faux. C'est pourtant l'erreur qui arrive dès qu'une classe hérite pour récupérer du code.

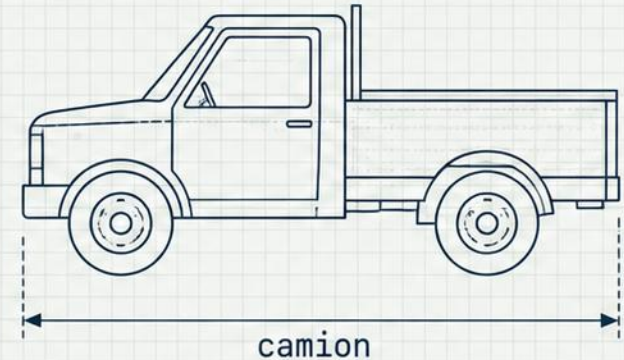
Si la phrase est vraie sans explication, l'héritage est une piste sérieuse. Vérifiez ensuite que les propriétés et les méthodes du parent conviennent vraiment à la classe fille.



# « Un camion, c'est une voiture en plus gros. » Vraiment ?

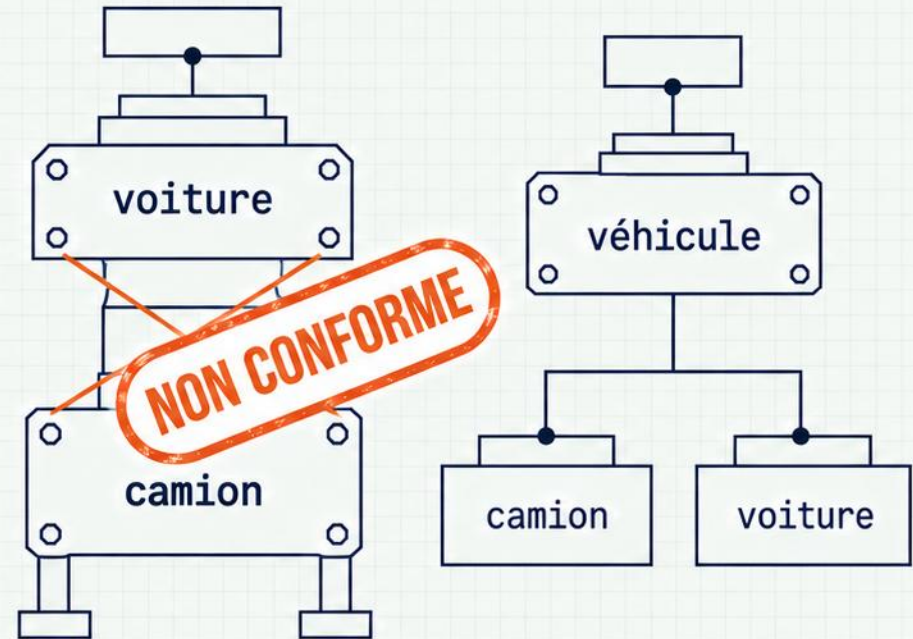
```
1 class Car
2 {
3     public function __construct(public readonly int $seats) {}
4
5     public function openTrunk(): string
6     {
7         return "Le coffre s'ouvre.";
8     }
9 }
10
11 // « Un camion, c'est une voiture en plus gros. » Vraiment ?
12 class Truck extends Car
13 {
14     public function load(float $tons): string
15     {
16         return "Chargement de {$tons} t.";
17     }
18 }
19
20 $truck = new Truck(seats: 5);
21 echo $truck->openTrunk(), PHP_EOL;
22 echo $truck->seats, ' places', PHP_EOL;
23
```

Le coffre s'ouvre.  
5 places



# Un camion n'a ni coffre ni cinq places : le résultat est absurde, PHP ne dit rien

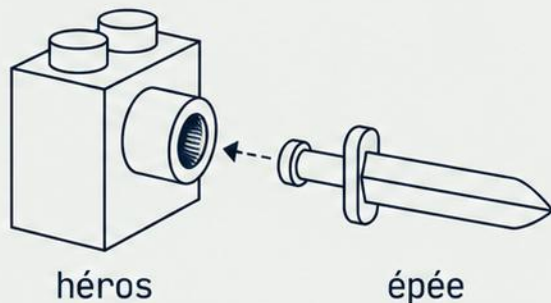
- Ces membres n'existent sur Truck que parce que Truck a hérité pour gagner du temps, et n'importe qui pourra les appeler.
- La phrase juste est « un camion **est un** véhicule », et « une voiture **est un** véhicule ». Le parent commun est Vehicle, pas Car.
- Ne choisissez donc pas l'héritage seulement pour réutiliser du code. La classe fille doit aussi représenter un cas particulier de la classe parente.



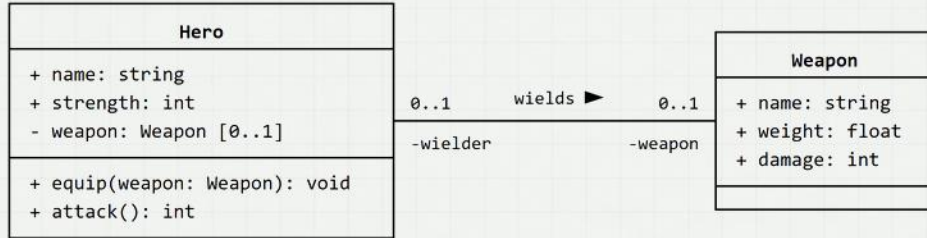
# Quand la phrase « est un » sonne faux, la réponse est presque toujours « a un »

```
1  final class Hero
2  {
3      /** hp, maxHp, isFullHealth, inventory, takeDamage(), heal(), isAlive() et __toString() :
4       * inchangés depuis les chapitres 3 et 5, omis ici pour ne montrer que l'arme. */
5      public private(set) ?Weapon $weapon = null;
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
17     public function equip(Weapon $weapon): void
18     {
19         $this->weapon = $weapon;
20     }
21
22     public function attack(): int
23     {
24         return $this->strength + ($this->weapon?->damage ?? 0);
25     }
26 }
27
28 $arthur = new Hero('Arthur');
29 printf('À mains nues : %d dégâts%s', $arthur->attack(), PHP_EOL);
30
31 $arthur->equip(new Weapon('Épée courte', 2.0, 5));
32 printf('Épée en main : %d dégâts%s', $arthur->attack(), PHP_EOL);
33
```

À mains nues : 2 dégâts  
Épée en main : 7 dégâts



# « Un X est un Y » donne extends, « un X a un Y » donne une propriété



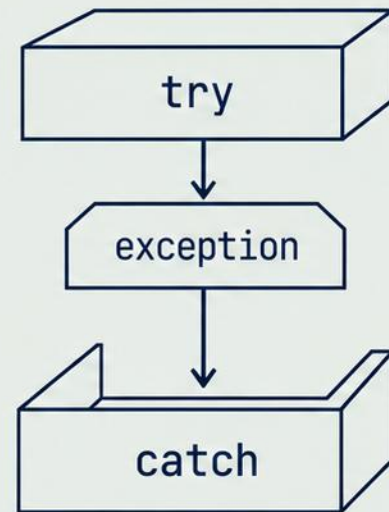
- « un X **est un** Y » : `class X extends Y`.
- « un X **a un** Y » : une propriété  
`public private(set) ?Y $y` dans X.
- L'arme arrive de l'extérieur, le héros peut en changer, et elle lui survit. Une classe fille, elle, ne change jamais de parent.
- Hero devient `final` : personne n'en hérite.

# Un try surveille un bloc, le catch du bon type reprend la main

Au chapitre 3, le constructeur de Dice lançait une `InvalidArgumentException` avec `throw`, et le programme s'arrêtait sur `Fatal error: Uncaught`. Voici ce qui se passe quand quelqu'un l'attrape.

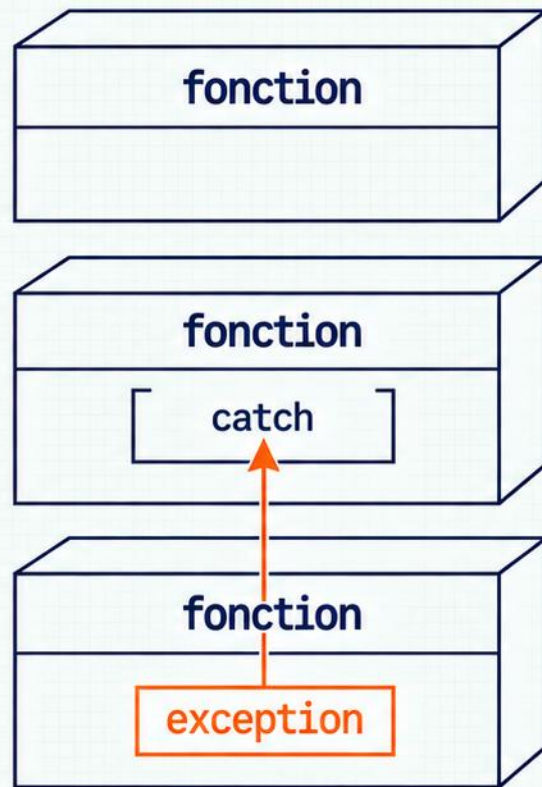
```
1 try {  
2     $dice = new Dice(0);  
3     echo 'Dé créé.', PHP_EOL;  
4 } catch (InvalidArgumentException $e) {  
5     echo 'Impossible : ', $e->getMessage(), PHP_EOL;  
6 }  
7  
8 echo 'Le programme continue.', PHP_EOL;
```

Impossible : Un dé a au moins 2 faces, 0 reçu.  
Le programme continue.



# Dès que throw s'exécute, PHP quitte le try à cette ligne

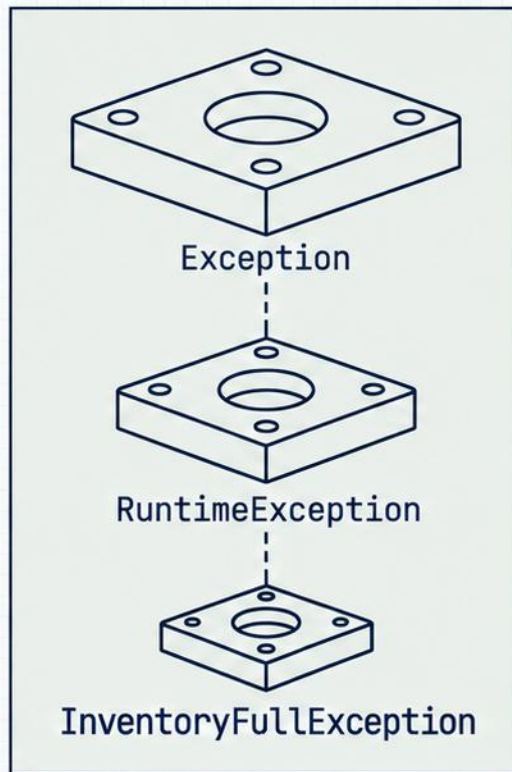
- Le bloc try est le code surveillé. new ne renvoie rien, \$dice n'existe pas, le echo 'Dé créé.' est sauté.
- PHP regarde ensuite les catch qui suivent. Le premier dont le type correspond à l'exception lancée, la classe elle-même ou une de ses parentes, reçoit l'objet dans \$e et s'exécute.
- Puis le programme reprend après le bloc, comme si rien ne s'était passé.
- Si aucun catch ne correspond, l'exception remonte à la fonction qui a appelé, puis à celle du dessus. Tout en haut sans être attrapée : Fatal error: Uncaught ..., et le programme s'arrête.



# Trois lignes, un corps vide : le nom de la classe rend le catch précis

```
1  /** Corps vide : tout vient de RuntimeException. Seul le nom compte. */  
2  final class InventoryFullException extends RuntimeException  
3  {  
4  }
```

- Les exceptions de PHP forment une hiérarchie de classes : Exception, puis RuntimeException pour ce qui ne se détecte qu'à l'exécution. Vous y ajoutez la vôtre avec extends.
- Les classes fournies par PHP vivent dans le **namespace global**, celui qui n'a pas de nom. Une barre oblique seule en tête d'un nom désigne ce namespace.
- Dans le dépôt du kata, vos fichiers commencent par namespace Dungeon; : écrivez extends \RuntimeException, sinon PHP cherche Dungeon\RuntimeException, qui n'existe pas.



# add() lève une exception au lieu de renvoyer false

```
1 public function add(Item $item): void
2 {
3     if ($this->totalWeight() + $item->weight > $this->maxWeight) {
4         throw new InventoryFullException(sprintf(
5             '"%s" ne rentre pas : le sac ne porte que %s kg.',
6             $item->name,
7             $this->maxWeight,
8         ));
9     }
10
11     $this->items[] = $item;
12 }
```

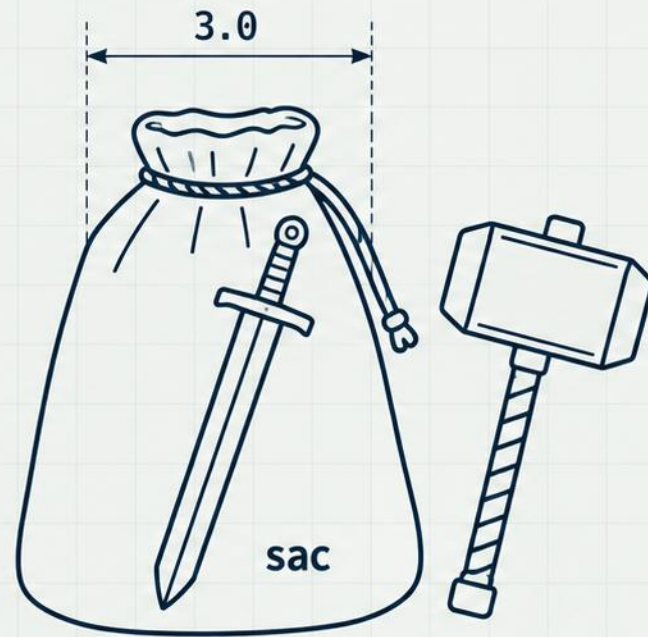


- Aux chapitres 4 et 5, add() était déclarée add(Item \$item): bool. Avec l'exception, le retour ne vaudrait plus jamais false, et une signature qui annonce un résultat impossible ment à celui qui la lit.
- Un false que personne ne teste disparaît en silence, alors qu'une exception non attrapée arrête le programme.

# Le try s'arrête à la ligne fautive, l'épée déjà ajoutée reste dans le sac

```
1  $inventory = new Inventory(maxWeight: 3.0);
2
3  try {
4      $inventory->add(new Weapon('Épée courte', 2.0, 5));
5      $inventory->add(new Weapon('Marteau de guerre', 12.0, 9));
6      echo 'Les deux armes sont dans le sac.', PHP_EOL;
7  } catch (InventoryFullException $e) {
8      echo 'Sac plein : ', $e->getMessage(), PHP_EOL;
9  }
10
11 echo 'Poids transporté : ', $inventory->totalWeight(), ' kg', PHP_EOL;
```

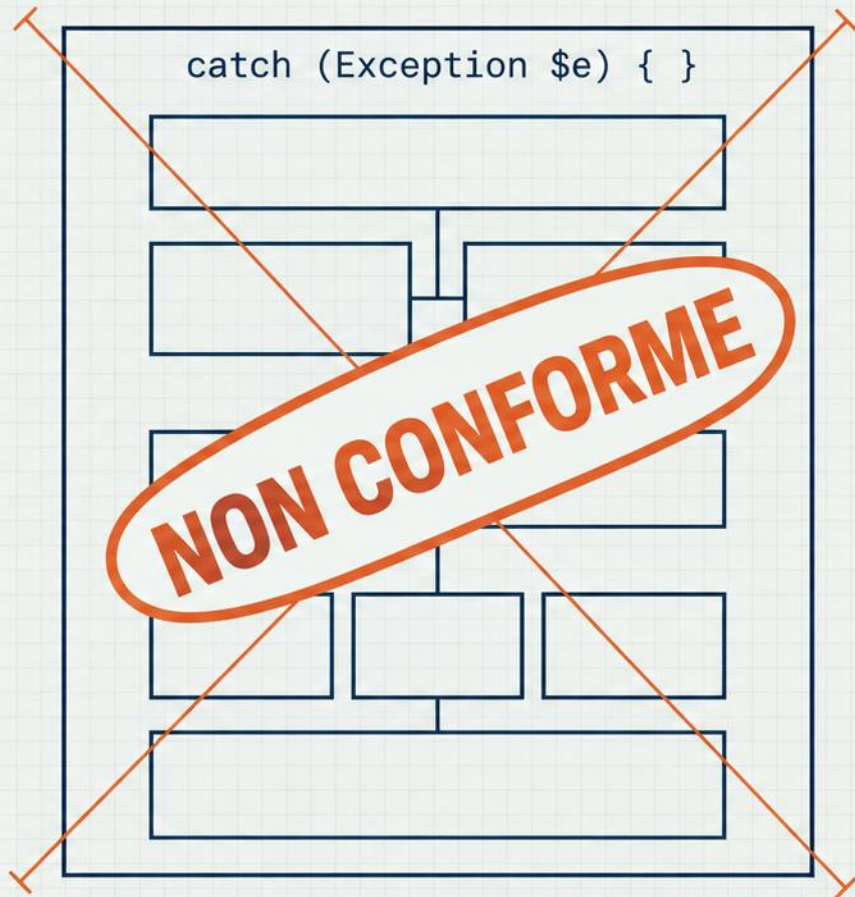
Sac plein : "Marteau de guerre" ne rentre pas : le sac ne porte que 3 kg.  
Poids transporté : 2 kg



Une exception n'annule pas ce qui a été fait avant. Et le catch est ciblé : il n'attrape que les sacs pleins. Une faute de frappe sur un nom de méthode dans le même bloc continue de provoquer une erreur.

# Attrapez ce que vous attendez, laissez passer le reste

- Attrapez ce que vous attendez, là où vous savez quoi faire : afficher un message au joueur, lui proposer de jeter un objet, réessayer. Le programme continue.
- Laissez passer ce que vous n'attendez pas : une exception qui n'aurait jamais dû arriver signale un bug, et le programme qui s'arrête avec sa trace vous dit où chercher.
- `catch (Exception $e) { }` autour de tout le programme cache les deux à la fois. Le joueur ne sait plus pourquoi son épée a disparu, et vous ne savez plus que le code est cassé.



# À vous : sur papier d'abord, puis le kata niveau 3

## Sur papier

1. Écrivez la phrase « est un » ou « a un » pour ces quatre paires, et dites ce que vous écririez en PHP :  
Dungeon/Room, Room/Monster, Potion/Item, Hero/Inventory.
2. `Inventory::remove()` ne fait rien quand l'objet n'est pas dans le sac. Faut-il lever une exception ? Justifiez en deux phrases. Restez sur papier : les tests du niveau 2 attendent que `remove()` ne fasse rien.

## Kata niveau 3

```
composer test -- --group=niveau-3
```

Item et Monster abstraites, Weapon, Potion, Goblin, Dragon, InventoryFullException, `Hero::equip()` et `Hero::drink()`. Lisez les tests avant de coder : ils sont l'énoncé complet.

